NORD PL

User's Guide

REVISION RECORD

Revision	Notes
09/75	Total revision, superseding all previous versions
10/76	Total revision, superseding all previous versions
07/77	Revision A. The following pages have been revised: 2–4, 3–1, 3–2, 3–3, 3–15, 3–17, 3–18, 3–19, 3–22, 3–27, 3–28, 7–2, A–1, A–3, and changed Index to Appendix E, and revised page vi.

NORD PL – User's Guide
Publ. No. ND-60.047.03

NORSK DATA A.S.
Lørenveien 57, Postboks 163 Økern, Oslo 5, Norway

PREFACE

This manual is recommended as a necessary documentation to any programmer/system analyst intending to obtain information about the NORD Programming Language (NORD PL). It contains the *definition of the language*. (The NORD PL compiler itself, however, is described in the manual NORD PL Program Documentation.)

According to the many examples and the consistent demonstration of the MAC Assembly Language equivalents of the different NORD PL statements, this manual is very well adapted to a self-study. It will also represent the main contents of the course US08 – NORD Programming Language arranged by the Educational Department of A/S Norsk Data-Elektronikk.

The operating system SINTRAN III of the NORD-10 computer family is written in NORD PL. This manual should, therefore, be given a closer consideration by any person wanting to become a system analyst or to attend the courses US04 – SINTRAN III and US05 – SINTRAN III Workshop.

It is also recommended that the reader of this manual should have attended the course US01 – MAC Assembly Language or at least have obtained some knowledge about MAC and especially about its addressing structure.

Finally, one important advice to the NORD PL programmer should be given: *NORD PL is not a problem oriented high level language like FORTRAN or COBOL. It is a machine oriented medium level language introduced to simplify the assembly coding, i.e., in any statement written the programmer should call attention to the influence on the register contents!*

TABLE OF CONTENTS

1 INTRODUCTION

Computer programming languages are divided into three classes: assembly languages, machine oriented languages and problem oriented languages. This manual defines the machine oriented language NORD Programming Language (abbreviation: NORD PL) running on the NORD-1 and NORD-10 computers.

1.1 *MACHINE ORIENTED LANGUAGES*

A machine oriented language is a medium level language standing between the problem oriented languages (high level languages) and assembly code. The syntax resembles that of ALGOL. However, the use is intended to be like that of an assembler, because all facilities of the computer can be reached:

1. The complete assembler instruction set with all addressing modes.
2. All registers.
3. All available memory location.

Comparing a machine oriented language to assembly code:

1. It is easier to write programs and the error checking can be more extensive.
2. The programs will be more readable for others.

Comparing a machine oriented language to high level language:

1. A PL language will give more optimal object code, about the same as for assembly code.
2. The programmer is not dependent on fixed calling sequences or data structures.

One of the main applications of machine oriented languages is system programming (operating systems, compilers), where efficiency as well as readability is needed.

1.2 *PROPERTIES OF THE NORD PL*

The NORD Programming Language is a language suitable for expressing a large class of computer system processes like operating systems, compilers and data networks. For instance, the SINTRAN III operating system, the FORTRAN compiler and even the NORD PL compiler itself is written in NORD PL.

Generally, it is recommended as an effective tool in the solution of almost any programming problem normally expressed in the MAC assembly language.

The NORD PL is designed for the NORD-1 and NORD-10 computers. The object output is MAC assembler source code. Therefore, including of assembly code sequences is very easy. All the debug facilities of MAC are immediately available, including symbolic references to labels and variables.

The statement set includes:

1. Declaration statements, with type specifications and data presetting.

2. Arithmetical statements, consisting of arithmetical/logical expressions and assignments. Constant expressions are also included. These will be evaluated at compile time.

3. Control statements, including:

GO	unconditional branching
IF	conditional branching
FOR	loop control
WHILE	conditional loop control
CALL	subroutine call

Input/Output statements are not available in the NORD Programming Language itself. Thus, I/O operations should be performed by means of MAC monitor calls introduced to the NORD PL program as normal assembler statements starting with an asterisk (*). (See Sections 2.4 and 4.3.) Additionally, the NORD PL compiler is supplied with a useful set of commands, which may be inserted at any point of a program. These commands inform the compiler about how to interpret the syntax of a program, where to put the object program, etc. They are described in Sections 4.1, 4.2 and 4.3.

The compiler also includes conditional compiling (see Section 4.2).

1.3 *FORMALISM FOR SYNTACTIC DESCRIPTION*

The syntax of the NORD Programming Language will be described with the aid of a metalinguistic formalism similar to the one used in the definition of COBOL.

The brackets { } show that one of the alternative statement parts given inside of these brackets should be chosen. A selection is obligatory.

The brackets [ ] show that the statement part given inside of these brackets is optional.

Terms enclosed in the brackets < > are meant to be self-explanatory and represent metalinguistic variables whose values are sequences of characters.

These three bracket types are not used in NORD PL itself.

The three dots . . . denote that repetitions of the last statement part are allowed.

Any mark in a formula which is not a metalinguistic variable, the three dots or one of these brackets, denote itself.

1.4 *ENVIRONMENTS*

The compiler needs about 6.5K of memory plus main symbol table (5 locations per symbol). The text is compiled in one pass.

The compiler can be run either as a freestanding system, under the TSS or under the SINTRAN III operating systems.

2 THE STRUCTURE OF NORD PL

2.1 *BASIC ELEMENTS*

NORD PL is built up from the following basic elements: identifiers, numbers, character and string constants, operators, delimiters and reserved symbols.

2.1.1 *Identifiers*

An identifier is a string of digits and letters, the first 5 characters only being significant. The rest will be regarded as a comment. At least one of the 5 first characters must be a letter (not necessarily the very first). An identifier may be used as the name of a variable, a label or a symbolic constant.

Example:

NEW, LOOP, INT2, 1A, 450 SLC, J2, 1976 SALARY

2.1.1.1 RESERVED IDENTIFIERS

Some identifiers are reserved for special use, as operators, statement symbols or register names. Some special characters are also used. A complete list is found in Appendix A.2. Refer also to Section 2.1.1.2.

2.1.1.2 REGISTERS

The registers have fixed names:

P	Program counter
X	Index register
T	T register
A	Accumulator
D	D register
L	Link register
B	Base register
AD	Double accumulator
TAD	Floating accumulator
K	One bit accumulator
Z	One bit floating point overflow
Q	One bit dynamic overflow
O	One bit static overflow
C	One bit carry
M	One bit multishift link
0	Zero register

If the number 0 is found in an executable expression, it will be regarded as the zero register, not as a constant.

2.1.2 *Constants*

There are four different types of constants: numbers or numerical constants, symbolic constants, character constants and strings.

2.1.2.1 NUMBERS

Numbers (also called numerical constants or constants) are either integers or floating point numbers. The compiler operates either in *decimal mode* or in *octal mode*. In decimal mode, the compiler will normally regard a string of digits as a decimal integer. If the string is immediately preceded by a &, it will be an octal integer. In octal mode the digit string will be octal in any case. Decimal/octal mode is set by commands (see Section 7.1). Initially, the compiler is in octal mode.

Floating point numbers have the same syntax as MAC floating point, except that the sign # is used instead of E; besides the number must always start with a digit.

Example:

0.3 # –33

2.1.2.2 CHARACTER AND STRING CONSTANTS

Character constants have the same syntax as in MAC. # # A puts the 7 bits ASCII equivalent of A right adjusted in a word, and # AB packs the characters A and B into one word. The string has also the same syntax as in MAC, i.e., it must be surrounded by simple quotation marks. The last quotation mark will be regarded as a part of the string.

Example:

'ABCD' will be packed as:

A	B
C	D
'	

2.1.2.3 SYMBOLIC CONSTANTS

Identifiers may be declared as symbolic constants to represent certain numerical or character values. Symbolic constants do not occupy any memory space at run-time. See also Section 3.1.3.

2.1.3 *Operators and Delimiters*

Operators consist of letters or of special characters. Delimiters are special characters with a particular significance. A complete list is found in Appendix A.

2.2 *DATA STRUCTURE*

Three data types are available.

1. Integers (16 bits)
2. Double (32 bits)
3. Triple (48 bits)

In addition, the data type Real may be used. This type is equivalent to Triple if 48 bits floating point format is used and equivalent to Double if 32 bits floating point format is used.

These types can be used either as single variables or arrays. Pointers to the actual variables and to arrays may also be declared.

All data must be declared before they can be used. However, the actual location can be delayed, allowing, for instance, a data table to be placed after the code using it.

Data locations or variables may get more than one identifier attached to it. (See the last part of Section 3.1.1.)

The addressing mode is defined by the context of the declaration.

Data locations may be initialized at compile time. Data may also be allocated at compile time without getting any identifier attached to it. (See Section 3.1.1.)

2.3 *DATA EXPRESSIONS*

A data expression is evaluated at compile time. The operands consist of constants and identifiers. If labels or variables are used, their address values will be used. The operators are:

+ Add
– Subtract
* Multiply
\ Byte separation (equivalent to "$*400_8$ + ")

The expression is evaluated strictly from left to right and all the arithmetic operators have the same priority. If a label or a reference of a variable occurs, only + and – are allowed for the rest of the expression.

Example:

NORD PL	MAC equivalents:
1 + 2*10	30
1\1	401
# # A \ # # B (equivalent to # AB)	# AB
5 + 2 + LAB1 – VAR2	7 + LAB1 – VAR2

Data expressions may appear in:

1. Declaration statements as initializations
2. Call statements as parameters
3. Executable statements as operands, surrounded by quotes (")

Indiced variables, i.e., array identifiers or array pointer identifiers followed by an index, included in paranthesis, may not appear in data expressions. An indexed variable, must be represented by the identifier combined with the operators + and – and proper constants. (See the last example in Section 3.2.3.)

2.4 *STATEMENT STRUCTURES*

A statement is normally terminated by a semicolon or a carriage return. Using the command @ICR, it is possible to set the compiler in "ignore-carriage return" mode, so that the carriage return will be ignored. Then, a statement can consist of several lines.

There is no use of parenthesis structure.

Example:

(The following might very well be a part of a NPL subroutine:)

```
A:=B=:X:=:D–T+P*VAR SHZ 4 SH "–2" SHR 10  SHL 1 ∧ 377 ∨D
XOR T BONE 3 BZERO 17
```

which is equivalent to:

```
COPY SB DA; COPY SA DX; SWAP SA DD; RSUB ST DA; RADD SP
DA; MPY VAR; SHA ZIN 4; SHA SHR 2; SHA ROT 10; SHA LIN 1;
AND (377; RORA SD DA; REXO ST DA; BSET ONE 30 DA;
BSET ZRO 170 DA
```

A comment begins with the percent sign (%). Then the rest of the line will be ignored.

If a statement starts with an asterisk (*), the rest of the line is regarded as MAC assembly code. These statements are passed on to the object output stream without the asterisk and without any error investigation.

If a statement starts with a circled alpha (@), it is regarded as a command to the compiler. See Sections 4.1, 4.2 and 4.3.

3 THE STATEMENTS OF NORD PL

The NORD PL statements are divided into two classes: declaration statements and executable statements.

3.1 *DECLARATION STATEMENTS*

Declarations serve to define certain properties of the quantities used in the program and to associate them with identifiers (names).

There are four types of declaration statements: data declarations, symbolic constant declarations, label declarations and subroutine declarations. With two exceptions mentioned in Section 3.1.4 and 3.1.5, declaration statements may occur everywhere in a program, the main rule being that the corresponding variables and symbols are referenced. Accordingly, it is a good rule to make those declarations as soon as possible, either before the subroutine declarations or immediately after a SUBR statement.

A symbolic name may not be declared twice, unless it appears as a local variable or a local label in different subroutines.

3.1.1 *Data Declarations*

Variables (data) should be declared to be one of the types:

INTEGER	1 word
DOUBLE	2 words
TRIPLE	3 words
REAL	2 or 3 words

Type declarations have the following general form:

where <expression> must follow the rules given in Section 3.3.

In addition, the two optional declaration symbols ARRAY and POINTER may be added.

The three basic data types may be elements of a one-dimensional array.

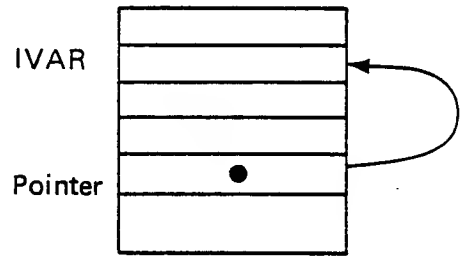
INTEGER ARRAY	(0 –∞ words)
DOUBLE ARRAY	(0 –∞ words)
REAL ARRAY	(0 –∞ words)

The array declaration has the following general syntax:

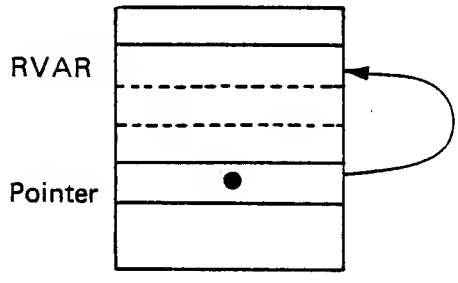
{INTEGER / DOUBLE / REAL / TRIPLE} ARRAY {<identifier> (<no. of elements>) / <identifier>:= (<list of expressions>) / <identifier>=<identifier> / <identifier>=?} [,{ }]...

Pointers to the variables may be declared:

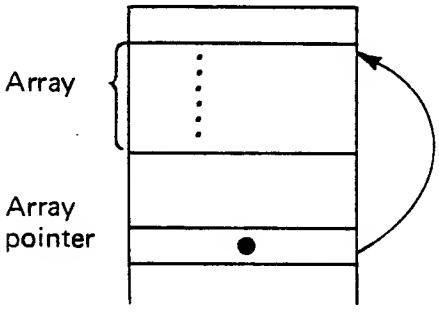
INTEGER POINTER (1 word)



DOUBLE POINTER (1 word)
REAL POINTER (1 word)



INTEGER ARRAY POINTER (1 word)
DOUBLE ARRAY POINTER (1 word)
REAL ARRAY POINTER (1 word)



As to be seen from the above figures, the contents of a pointer is the address of the first word of the variable to be pointed at.

A pointer occupies one word (16 bits) regardless of which type of variable it points at. The different pointer declarations INTEGER POINTER, DOUBLE POINTER, etc., describe the type of the variable addressed by the pointer.

The syntax is the following:

```
{INTEGER }
{DOUBLE  } [ARRAY]  POINTER {<identifier>                    }
{REAL    }                  {<identifier>:=<expression>      } [,{ } ]...
{TRIPLE  }                  {<identifier>=<identifier>       }
                            {<identifier>=?                  }
```

(Note: Pointer arrays do not exist in NORD PL.)

The different declaration types will modify the addressing mode. After the declaration symbols, a list of variables can follow. The variables can be initialized, either as default zeros, or to specified values.

Example:

	MAC Equivalents:
INTEGER INT1, INT2	INT1, 0 INT2, 0
INTEGER TRE:=3	TRE, 3
REAL FLX	FLX,0;0;0;
REAL PI:=3.1415	PI, [3.1415
DOUBLE SYM, SY2=SYM,S3	SYM,0;0 SY2=SYM S3,0;0
INTEGER POINTER PVAR:=VAR	PVAR,VAR
INTEGER ARRAY [ARR (12)	IARR=*; *+12/
REAL ARRAY FX(20), FY(30)	FX=*;*+20+20+20/ FY=*;*+30+30+30/
INTEGER ARRAY TEXT:= 'STRING'	TEXT, 'STRING'

Note the different meaning of the two symbols = and := in declarations.
= is giving the left side variable the same address as the right side variable.
:= is giving the value on the right side to the left side variable.

A declared entity may be initialized by several elements, divided by a comma, the whole list enclosed in parentheses:

```
INTEGER ARRAY AA:=(TRE,PI          AA,TRE
+ 1, 15)                           PI + 1
INTEGER ARRAY PARLIST:=            15
(LOGNO, AREA, "100", "15")         PARLI, LOGNO
                                   AREA
                                   (100
                                   (15
```

Each element may be a data expression.

Data may also be initialized with no name attached to it by using the DATA statement:

Example:

```
DATA (44,SY,5)
```

is equivalent to the three MAC statements: 44;SY;5.

The syntax is the following:

DATA (<list of expressions>)

where <list of expressions> contains expressions following the rules in Section 2.3 and separated by commas.

The actual allocation of data may be *delayed,* so that the data are placed after the code physically. Then, the variables are declared equal to *question mark* the first time and later they are declared a second time.

Example:

```
INTEGER ARRAY TABLE = ?
SUBR S
:
:
RBUS
INTEGER ARRAY TABLE (1000)
```

3.1.2 *Addressing Mode Specifications*

The addressing mode of a variable is dependent on the context of the declaration statement. If nothing else is stated, a variable declared inside a subroutine (between a SUBR and RBUS statement) is directly P-relative addressed. If declared outside of a subroutine, then it will be indirectly addressed. If a variable is to be B-relative, its declaration statement must be enclosed by a BASE-ESAB pair or a DISP-PSID pair.

All variables have an attribute determining the addressing mode. One and only one of the following attributes may be chosen for each variable: Global, Local, Base or Disp.

Global: Variables declared outside of SUBR-RBUS, BASE-ESAB and DISP-PSID. Global variables are indirectly P-relative addressed. Pointers may be declared as global variables, but then only their contents may be accessed. It is not possible to access another variable through a global pointer because of the resulting "double indirect" addressing mode.

Local: Variables declared inside of SUBR-RBUS but outside of BASE-ESAB and DISP-PSID. Local variables are directly P-relative addressed.

Base: Variables declared inside of BASE-ESAB. Base variables are B-relative addressed. (It is also possible to make them X-relative addressed.)

Disp: Variables declared inside of DISP-PSID. Disp variables are B-relative addressed. (It is also possible to make them X-relative addressed.)

In addition to the addressing attributes all identifiers have scope attributes informing about where they are defined, i.e., where they may be referred to. In respect to the scope attribute, all identifiers are either global or local, thus, even base and disp variables, as well as symbolic constants and labels, may be defined to be known globally or locally.

3.1.2.1 BASE VARIABLES

If the variables are static allocated, BASE may be used, followed by a base-field identifier (a label).

The declaration of base variables has the following general form:

BASE <base-field identifier> <data declarations> ESAB

Example:

```
                                    MAC equivalents:

BASE BA                             BA=*+200
    INTEGER BVAR1,BVAR2             BVAR1,0
                                    BVAR2,0
    INTEGER POINTER PSUB:=SUB       PSUB, SUB
ESAB
```

Base variables are normally B-relatively addressed.

Before accessing Base variables, the user must load the B-register with the value of the corresponding base field identifier. This may be done as follows:

```
A:="BA" =:B
```

The instruction to get a BASE variable may be:

```
A:=BVAR1                                LDA BVAR1 – BA,B
```

3.1.2.2 DISP VARIABLES

If the variables are dynamic allocated, for instance, as variables in an element of a data structure, DISP should be used. *DISP variables may not be initialized.*

The declaration of disp variables has the following general form:

DISP { <displacement> / <disp-field identifier>[=<displacement>] } <data declarations>PSID

The most usual way of declaring DISP variables is as follows:

Example:

```
                                        MAC equivalents:

DISP – 200
    INTEGER D1, D2                      D1=–200;D2=–177
    INTEGER ARRAY DARR (10)             DARR=–176
    INTEGER ENDA                        ENDA=–166
PSID
```

The following example illustrates how the disp field identifier works.

Example:

```
DISP DV = 10
    INTEGER NILS, PER
PSID
```

This statement sequence will define NILS=10 and PER=11. The PSID statement will change the value of DV (by compile time) from 10 to 12, thus, if one now says

```
DISP DV
    INTEGER EVA, BERIT
PSID
```

the definitions EVA = 12 and BERIT = 13 will be made. The last PSID statement will change the value of DV from 12 to 14.

In this way, it is possible to "continue" a DISP field later on in the program. This may be useful if one wants to put the local variables of different routines into one global DISP field.

DISP variables are normally accessed through the B-register. It is the user's responsibility to set the B-register to the proper value.

Note that DISP variables in contradiction to GLOBAL, LOCAL and BASE variables do not reserve any locations in the memory at compile time. They are only used as symbols in the displacement part of an address.

Example:

```
                                                MAC equivalents:
      INTEGER A,B                 } global      A,0
      INTEGER ARRAY C(10)         }             B,0
      BASE BA                                   C = *
          INTEGER D               } base        * +10/
          INTEGER POINTER E       }             BA = * +200
      ESAB                                      D,0
      SUBR RUT                                  E,0
      REAL F                      } local       F,0;0;0;
      REAL ARRAY G(5)             }             G = *
      DISP -200                                 * +17/
          DOUBLE H                } disp        H = -200
          INTEGER I               }             I = -176
      PSID
RUT:  =
      =
      =
      RBUS
```

Examples of the addressing of the global, local, base and disp variables:

Global:

```
      INTEGER M1,M2,RES
      SUBR MUL
MUL:  M1 * M2 =: RES
      EXIT
      RBUS
```

MAC equivalents:

```
      M1,0
      M2,0
      RES,0
MUL,  LDA I (M1
      MPY I (M2
      STA I (RES
      EXIT
)FILL
```

Local:

```
      SUBR MUL
      INTEGER M1,M2,RES
MUL:  M1 * M2 =: RES
      EXIT
      RBUS
```

```
      M1,0 ; M2,0 ; RES,0
MUL,  LDA M1
      MPY M2
      STA RES
      EXIT
)FILL
)KILL  M1  M2  RES
```

Base:

```
      SUBR MUL
      BASE BA
          INTEGER M1,M2,RES
      ESAB
MUL:  "BA" =: B
      M1 * M2 =: RES
      EXIT
      RBUS
```

```
      BA = * +200
      M1,0
      M2,0
      RES,0
MUL,  LDA   (BA
      COPY  SA DB
      LDA   M1 – BA,B
      MPY   M2 – BA,B
      STA   RES – BA,B
      EXIT
)FILL
)KILL  M1  M2  RES
```

Disp:
The calling subroutines define the B–register:

```
      SUBR MUL
      DISP 0
          INTEGER M1,M2,RES
      PSID
MUL:  M1 * M2 =: RES
      EXIT
      RBUS
```

```
      M1=0
      M2=1
      RES=2
MUL,  LDA M1,B
      MPY M2,B
      STA RES,B
      EXIT
)FILL
)KILL  M1  M2  RES
```

For a closer study of the addressing modes, consider Table 3.1.

ADDRESSING MODES:

		BASE	DISP	LOCAL	GLOBAL
INTEGER K	A:=K A:="K"	LDA K–BA, B LDA (K	LDA K, B LDA (K	LDA K LDA (K	LDA I (K LDA (K
INTEGER ARRAY K (10)	A:=K(X) A:="K"	LDA K–BA, B, X LDA (K	LDA K, B, X LDA (K	LDA I, X (K LDA (K	LDA I, X (K LDA (K
INTEGER POINTER K	A:=K A:="K"	LDA I K–BA, B LDA K–BA, B	LDA I K, B LDA K, B	LDA I K LDA K	Illegal LDA I (K
INTEGER ARRAY POINTER K	A:=K(X) A:="K"	LDA I K–BA, B, X LDA K–BA, B	LDA I K, B, X LDA K, B	LDA I K, X LDA K	Illegal LDA I (K
INTEGER CONSTANT (−128≤k≤127)	A:=4 A:="4"			SAA 4 SAA 4	
INTEGER CONSTANT (k < −128 ∨k >127)	A:=1000 A:="1000"			LDA (1000 LDA (1000	
DOUBLE K	AD:=K A:="K"	LDD K–BA,B LDA (K	LDD K, B LDA (K	LDD K LDA (K	LDD I (K LDA (K
DOUBLE ARRAY K(10)	AD:=K(X) A:="K"	LDD K–BA, B, X LDA (K	LDD K, B, X LDA (K	LDD I, X (K LDA (K	LDD I, X (K LDA (K
DOUBLE POINTER K	AD:=K A:="K"	LDD I K–BA, B LDA K–BA, B	LDA I K, B LDA K, B	LDD I K LDA K	Illegal LDA I (K
DOUBLE ARRAY POINTER K	AD:=K(X) A:="K"	LDD I K–BA, B, X LDA K–BA, B	LDD I K, B, X LDA K, B	LDD I K, X LDA K	Illegal LDA I (K
REAL K	TAD:=K A:="K"	LDF K–BA, B LDA (K	LDF K, B LDA (K	LDF K LDA (K	LDF I (K LDA (K
REAL ARRAY K(10)	TAD:=K(X) A:="K"	LDF K–BA, B, X LDA (K	LDF K, B, X LDA (K	LSF I, X (K LDA (K	LDF I, X (K LDA (K
REAL POINTER K	TAD:=K(X) A:="K"	LDF I K–BA, B LDA K–BA, B	LDF I K, B LDA K, B	LDF I K LDA K	Illegal LDA I (K
REAL ARRAY POINTER K	TAD:=K(X) A:="K"	LDF I K–BA, B, X LDA K–BA, B	LDF I K, B, X LDA K, B	LDF I K, X LDA K	Illegal LDA I (K

Table 3.1.

Table 3.1, continued.

		BASE	DISP	LOCAL	GLOBAL
REAL CONSTANT	TAD:=3.14 TAD:="3.14"			LDF ([3.14 LDF ([3.14	
LABEL (K:)	GO K A:="K"			JMP K LDA (K	JMP I (K LDA (K

3.1.3 *Symbolic Constants*

Identifiers may be defined to be symbolic constants, using the SYMBOL declaration statement, which has the general format:

SYMBOL { <identifier> / <identifier>=<expression> } [,] ...

where <expression> may be a number within the interval [-200_8, + 177_8], a character constant, a symbolic constant already defined, or another data expression following the rules described in Section 2.3.

An <identifier> listed in a SYMBOL declaration statement will adopt the value evaluated in the corresponding <expression>. An <expression> is evaluated at compile time.

If an <expression> is not given for the first <identifier> in the list, the value zero is assigned to it. Other <identifier>s without a corresponding <expression> will represent a value 1 greater than the preceding value.

Example:

	MAC equivalents:
SYMBOL L200=200, L210=L200+10	L200=200 L210=L200+10

A SYMBOL declaration does not allocate any memory space. It is not possible to change the value of a symbolic constant during run-time.

Example:

	MAC equivalents:
SYMBOL S0, S1, S2, S3	S0 = 0 S1 = 1 S2 = 2 S3 = 3

SYMBOL CHA = # # A, CHB, CHC	CHA = # # A
	CHB = CHA + 1
	CHC = CHB + 1

3.1.4 *Label Declarations*

A label is an identifier (see Section 2.1.1) used as a name of a location in the memory containing an executable statement. A label declaration has the general form:

<label>:[<executable statement>]

Label declarations may only occur inside of subroutines.

There are two types of labels; entry points and local labels.

Entry points are global and may be referenced from outside of the subroutine where they are declared. Entry points have to appear in a SUBR statement and may only be declared once in the same program.

Local labels will be killed at the end of the subroutine in which they are declared and may, thus, only be referred to from inside that subroutine. An identifier may be declared as a local label in more than one subroutine without confusion, but only once within one actual subroutine.

Labels may be referenced in GO, GOSW and CALL statements.

3.1.5 *Subroutine Declarations*

A subroutine statement starts with the symbol SUBR, followed by a list of entry points. The entry points will be global labels. Other labels and variables declared after the subroutine heading will be killed at the end of the subroutine. The end is marked by the symbol RBUS.

The RBUS statement provides the two MAC assembler commands)FILL and)KILL.

Example:

```
SUBR ENT1, ENT2
:
:
ENT1:
:
:
ENT2:
:
:
EXIT
RBUS
```

It is the programmer's responsibility to provide a return jump from the subroutine (using EXIT, EXITA or GO).

Example: (Saving of the link register in subroutines.)

```
        SUBR RUT
        INTEGER POINTER RETUR
RUT:    A:=L=:"RETUR"          COPY SL DA
        :                      STA RETUR
        :
        :
        GO RETUR               JMP I RETUR
        RBUS
```

There is only one level of subroutine declarations, i.e., a subroutine cannot be declared inside of another subroutine.

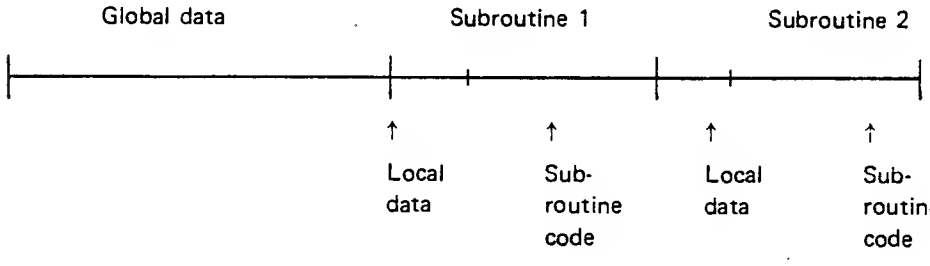
It is, however, possible to call subroutines from another subroutine and in this manner to construct a nesting of subroutine calls up to any level wanted. (See also Section 3.2.5.2.)

3.1.6 *Program Structure*

Since the NORD-1/10 computers have direct addressing areas of 256 words, the programs will usually be divided into small subroutines. Therefore, NORD PL has a subroutine feature, where labels and variables defined outside the subroutine are global.

There is only one level of subroutine declarations; it is not possible to declare a subroutine within another subroutine.

Example:



3.2 *EXECUTABLE STATEMENTS*

Executable statements are divided into arithmetical statements and control statements.

An executable statement can only appear within a subroutine. This means that any NORD PL program should contain at least one subroutine. The entry point of this "main" subroutine is then considered the starting point of the program.

In general an executable statement specifies a series of operations between the *primary operand,* which is a register, and different *secondary operands,* which can be registers, variables or constants. If the expression starts with a register, this register will be the primary operand throughout the expression.

Example:

NORD PL	MAC equivalents:
A+V1+55=:V2	ADD V1 AAA 55 STA V2
X+5=:L–10	AAX 5 COPY SX DL AAAX –10

In the first example above the A register is the primary operand, the identifiers V1, V2 and the numerical constant 55 are secondary operands. In the second example the X register is the primary operand, the L register and the numerical constants 5 and 10 are secondary operands.

The operations are executed strictly from left to right, with no implicit priority.

3.2.1 *Operators*

Arithmetical:

:=	Load
=:	Store
:=:	Swap
–	Subtract
+	Add
*	Multiply
/	Divide (reals only)

Shift:

SHZ	Shift with zero end input
SH	arithmetical shift
SHR	rotational shift
SHL	shift with link end input (bits shifted into the register are taken from the M bit in the status register, bits shifted out are fed to M. This corresponds to an extended 17 bits rotational shift.)

Logical:

/ \	And
\ /	Or
XOR	Exclusive or
–,	One's complement
–	Two's complement
BONE	Set bit to one
BZERO	Set bit to zero

Special Unary:

MIN	Memory increment (MIN instruction)
GOSW	Switch

Example:

NORD PL	MAC equivalents:
A+4 GOSW L1, L2	AAA 4 RADD SA DP JMP L1 JMP L2
T SHZ – 2	SHT ZIN SHR 2

A NORD PL statement (or expression) may not start with any operator except for the MIN and GOSW operators.

An expression may also start with a constant, a symbolic constant or a variable. *Then the A, AD or TAD register will be the primary operand,* if the variable is an integer, double or triple. The first operation is then assumed to be a load.

Example:

If IX, IY and IZ are integers, then the expression: IX + IY = :IZ is equivalent to A:=IX + IY = :IZ.

NORD PL	MAC equivalents:
IX + IY = :IZ	LDA IX ADD IY STA IZ
IX SHR 2	LDA IX SHA ROT 2

The NORD Programming Language is to be considered as a convenient simplification of the MAC assembler language. Therefore, generally the same rules which in MAC apply to the use of registers and to all the operations mentioned in this section, also constitute severe restrictions in NORD PL.

Two operators may normally not appear next to another without an operand inbetween them. For exceptions, see Sections 3.2.1.1 and 3.2.1.3.

3.2.1.1 ARITHMETICAL OPERATIONS

The arithmetical operations have the general form:

[<primary operand>] <operator> <secondary operand>

The <primary operand> may be omitted when it is given by a previous operation in the same statement.

The := (load) operator may have as its <primary operand> any register mentioned in Section 2.1.1.2 except for the zero register. When the <primary operand> is a 1 bit register only the constants 1, "0", "1", or a symbolic constant which is set equal to 1, or the same symbolic constant in quotes are allowed as <secondary operand>.

When the <primary operand> is the A, B, T or X register, the <secondary operand> may be a constant or a symbolic constant (see Section 3.1.3) or a quoted expression. The B register only accepts values within the range of −200, +177.

The =:(store) operator may have as its <primary operand> any register mentioned in Section 2.1.1.2, except for the 1 bit registers.

The loading/storing of a 16 bit register into/from any other 16 bit register is legal and will be compiled into a COPY instruction.

An integer variable may only be loaded/stored from/into a memory location declared as integer into/from the A register.

A double variable may only be loaded/stored from/into a memory location declared as double (or real if 32 bits floating point is used) into/from the AD register.

A triple variable may only be loaded/stored from/into a memory location declared as triple (or read if 48 bits floating point is used) into/from the TAD register.

The :=: (swap) operator may only have 16 bit registers as <primary operand> and <secondary operand>.

The + (add) and – (subtract) operations may only be used on integer and triple (real) variables if 48 bits floating point format is used, and only on integer and double (real) variables if 32 bits floating point format is used. When the D, L or P register is the <primary operand>, the <secondary operand> should be a register. When the B, T or X register is the <primary operand>, the <secondary operand> should be a register, a constant or a symbolic constant or a quoted expression with a value with the interval –200, +177.

It is necessary to distinguish between the different interpretations of the character –. Primarily, this character is interpreted as the subtrack operator, which is normally compiled into a SUB instruction. (may also become an AAA, AAT, AAX or a RSUB instruction.) But the character – also denotes the negative sign of a numerical or symbolic constant. This case represents an exception in which the character – is allowed to appear directly after a := operator (e.g. A:=–5).

The third interpretation of the character – is explained in Section 3.2.1.3.

The character + may only be used as the sign of a constant, of a symbolic constant or of a variable when it appears inside of quotes (e.g. T SH "+2", see Section 3.2.2). Otherwise, it is always interpreted as the add operator which is normally compiled into the ADD instruction. (May also become an AAA, AAT, AAX or RADD instruction.)

A * (multiply) operation may only be performed on constants, symbolic constants, integer and real variables. The multiply operation does not apply to registers. It is compiled into the MPY or FMU instructions. The RMPY instruction is only available as a MAC statement starting with an asterisk.

A / (divide) operation only applies to real variables. It is compiled into the FDV instruction. The RDIV instruction is only available as a MAC statement starting with an asterisk.

3.2.1.2 SHIFT OPERATIONS

The shift operations have the general form:

[<primary operand>] <operator> <number of shifts>

<primary operand> is restricted to the A, D, T and AD registers and may be omitted when the register to be shifted is given by some previous operation in the same statement (see examples in Section 3.2.1).

<number of shifts> may be a constant, a symbolic constant or a quoted expression. A positive value will give left shifts, a negative value will give shifts to the right.

3.2.1.3 LOGICAL OPERATIONS

The logical operations / \ (and), \ / (or) and XOR (exclusive or) have the general form:

[<primary operand>] <operator> <secondary operand>

The <primary operand> may be any 16 bit register and may be omitted if it is given by some previous operation in the same statement. <secondary operand> may be any 16 bit register. These operations are then compiled into the RAND, RORA and REXO instructions.

If the A register is the <primary operand>, the / \ and \ / operators may also have a constant, a symbolic constant, a quoted expression or an integer variable as the <secondary operand>. They are then compiled into the AND and ORA instruction.

The –, (one's complement) and the – (two's complement) operators have the general form:

[<primary operand>] <operand>

The operators are unary, i.e., they only have one operand that always is a register. This <primary operand> may be omitted when it is given by some previous operation in the same statement. These operators may proceed or succeed other operators.

Example:

NORD PL	MAC equivalents:
A–	COPY DA SA CM1
A–+T	COPY DA SA CM2 RADD ST DA
A–12––,:=–5	AAA –12 COPY DA SA CM2 COPY DA SA CM1 SAA 177773

The BONE (set bit) and BZERO (clear bit) operations have the general form:

[<primary operand>] <operator> <bit number>

The <primary operand> may be any 16 bit register and may be omitted when it is given by some previous operation in the same statement. <bit number> may be a constant, a symbolic constant or a quoted expression.

3.2.1.4 THE MIN AND GOSW OPERATORS

The MIN operator has the general form:

MIN <integer variable>

and is compiled into the MIN instruction. The <integer variable> may be any variable declared as integer. The MIN operator increases the value of the <integer variable> by 1. If the result is zero, the next instruction is skipped, otherwise, it is executed.

The GOSW operator has the general form:

[<primary operand>] GOSW <list of labels>

The <primary operand> may be any 16 bit register and may be omitted.

If a <primary operand> is not given by a previous operation in the same statement, the A register is considered as default register. <list of labels> is the names of any number of labels separated by commas.

The operation is compiled into a RADD instruction with the primary operand as source register and the P register as destination register, followed by a number of JMP instructions corresponding to the labels given in the list.

Example:

NORD PL	MAC equivalents:
MIN IVAR	MIN IVAR
GOSW L1, L2, L3	RADD SA DP JMP L1 JMP L2 JMP L3
X GOSW L1, L2	RADD SX DP JMP L1 JMP L2
GOSW FAR L1, FAR L2	RADD SA DP JMP I (L1 JMP I (L2

3.2.2 *Double Quotation Marks*

If a variable is included in quotes ("), it is said to be *referenced* . Then, it is accessed one level less indirect than otherwise. This means that a referenced pointer will be accessed as a variable, and a referenced variable or label will give the address value. A constant will have the same meaning whether it is referenced or not.

Inside the quotes even whole data expressions can be placed (see Section 2.3). Then all variables will be represented with their address values (even the pointers). Quoted expressions are evaluated at compile time.

Example:

NORD PL	MAC equivalents:
INTEGER TRE:=3	
INTEGER POINTER PP:=TRE	
A:="3"	SAA 3
A:=3	SAA 3
A:="TRE"	LDA (TRE
A:=TRE	LDA TRE
A:="PP"	LDA PP
A:=PP	LDA I PP

Example

Let PNT1 be an integer pointer and V1 an integer variable.

NORD PL	MAC equivalents:
PNT1	LDA I PNT1
"PNT1"	LDA PNT1
"PNT1+0"	LDA (PNT1+0
V1	LDA V1
"V1"	LDA (V1
"PNT1–V1+5"	LDA (PNT1–V1+5

The number 0 will be referred to as the zero register. If a constant with value zero is wanted, it should be surrounded by quotes (").

Example:

NORD PL	MAC equivalents:
0=:T	COPY DT
0=:VAR	STZ VAR
"0"=:VAR	SAA 0; STA VAR
A BONE "0"	BSET ONE 0 DA
A SHZ 2 + 5	SHA ZIN 2 AAA 5
A SHZ "2 + 5"	SHA ZIN 7

3.2.3 *Reference of Elements of any Array*

Elements of an array are referred to by the name of the array, followed by a left parenthesis followed by an expression followed by a right parenthesis:

ARRAYNAME <(expression)>

The <expression> may consist of the name of a 16 bit register, an integer variable, an integer pointer, a symbolic constant, a numerical constant or a quoted expression. No operators (+, –, * or /) are allowed inside the parenthesis except when the expression is surrounded by double quotation marks. A constant should, therefore, be positive or zero without sign, but the value referred to by an integer variable, an integer pointer or a symbol might as well be negative.

Example:

Let ARR be an array, IVAR an integer, IPOINT an integer pointer and SYMB a symbolic constant. The following statements are now legal:

```
ARR (A)
ARR (X)
ARR (IVAR)
ARR (IPOINT)
ARR (SYMB)
ARR (5)
ARR ("2+2–SYMB")
ARR ("IVAR")
```

The X-register is always used to keep the value of the index, i.e. the expression inside the parenthesis is evaluated and loaded to the X-register.

There is no control whether the value of the index given is ranging outside of the declared number of elements in the array.

If the array is declared as TRIPLE ARRAY or DOUBLE ARRAY 3 or 2 words respectively are loaded starting with the 16 bit word referred to by the index. The programmer is himself responsible for making the index point to a proper location:

Example:

If the first element of the array declared as TRIPLE ARRAY ARR (1000) is wanted, a correct reference is ARR (0). The second element could be referred to by ARR (3), the third by ARR (6) and the fourth by ARR (11).

Elements of an array may also be accessed through an array pointer. Consider the following example:

```
TRIPLE ARRAY RA1 (5)
TRIPLE ARRAY POINTER RAP1:=RA1, RAP2
"RA1 + 3" =: "RAP2"
TAD:=RAP1 (0) + RA 1 (3) =: RAP2 (11)
```

The two first elements of the triple array RA1 will be added together and stored into the last (fifth) element. The last statement in the above example is equivalent to RA1 (0) + RA1 (3) =:RA1 (14)

Note that when referring to an array through an array pointer, *the appropriate index of the array pointed at* must be included in parenthesis and succeed the array pointer identifier. The syntax in this case is just the same as when using the array identifier itself.

3.2.4 *X-Relative Addressing*

Variables declared in a DISP- or BASE-field (and only those) can be forced to be X-relatively addressed instead of B-relatively addressed. Then, the variable must be preceded by an X value denotation and a period (.).

Example:

NORD PL	MAC equivalents:
INTEGER START	START, 0
DISP 5	DD = 5
INTEGER DD, EL1, VAL	EL1 = 6
PSID	VAL = 7
X.DD	LDA DD,X
START.DD	LDX START LDA DD,X

START.EL1.VAL	LDX START
	LDX EL1,X
	LDA VAL,X

This access method is useful for addressing data structures. Starting at the first period, the value represented by the identifier preceding the period is loaded to the X register. Values represented by identifiers between two periods are successively loaded into the X-register, using X-relative addressing. The value represented by the last identifier is then loaded into the primary register using X-relative addressing. There is no limit for the number of periods in one such expression.

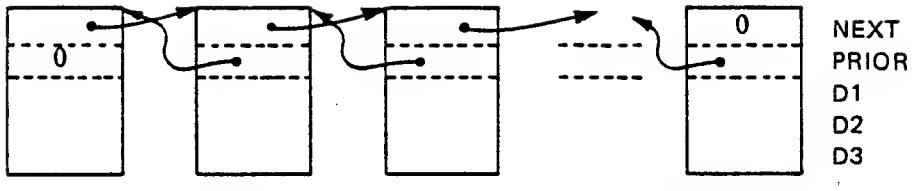
Integer pointers may only occur before the first period. Other pointers are not allowed unless they are "quoted". Indexed variables, i.e., arrays or array pointers succeeded by an index included in paranthesis may not appear in an expression using X-relative addressing.

Double and real variables may only appear after the last period.

Register names may only occur before the first period. Quoted expressions are legal.

Examples:

Consider the following list of data elements each containing the five integers NEXT, PRIOR, D1, D2 and D3.




```
NORD PL                                  MAC equivalents:
% SUBROUTINE FINDING THE
% WORD D1 IN THE ELEMENT
% POINTED AT BY THE
% A-REGISTER
% D1 IS RETURNED IN THE
% T-REGISTER
DISP 0
  INTEGER NEXT,PRIOR;D1,D2,D3
                                         NEXT=0 ; PRIOR = 1 ;
PSID                                     D1=2 ; D2=3 ; D3=4 ;
SUBR RUT1
RUT1: T:=A.D1                  RUT1,     COPY SA DX
EXIT                                     LDT D1,X
RBUS                                     EXIT

% SUBROUTINE FINDING D3 IN THE NEXT ELEMENT
SUBR RUT2
RUT2: T:=A.NEXT.D3             RUT2,     COPY SA DX
EXIT                                     LDX NEXT ,X
RBUS                                     LDT D3, X
                                         EXIT

%SUBROUTINE FINDING D3 IN TWO
% ELEMENTS IN FRONT OF THE ELEMENT
% POINTED AT BY THE A-REGISTER:
SUBR RUT3
RUT3: T:=A.PRIOR.PRIOR.D3 RUT3,          COPY SA DX
EXIT                                     LDX PRIOR,X
RBUS                                     LDX PRIOR,X
                                         LDT D3,X
                                         EXIT

% SUBROUTINE FINDING D2 IN THE
% LAST ELEMENT OF THE LIST. THE
% A-REGISTER IS INITIALLY POINTING
% TO A CASUAL ELEMENT.
SUBR RUT4
RUT4: IF A.NEXT > <0 GO RUT4
T:=X.D2
EXIT
RBUS
```

3.2.5 *Control Statements*

The control statements include unconditional and conditional branching, unconditional and conditional loop control and subroutine calls.

3.2.5.1 UNCONDITIONAL BRANCH STATEMENTS

The GO statement is used for unconditional branching. It consists of the symbol GO followed by a label or a pointer.

Examples:

```
                                        MAC equivalents:

INTEGER POINTER RET:=RETX                   RET, RETX
LL: GO RET                            LL,   JMP I RET

    GO LL                                   JMP LL

    GO ENTX  % EXTERNAL                     JMP I (ENTX
             % ENTRY POINT
```

If a subroutine is long, it can be useful to force a jump to be indirect, as not to exceed the displacement range. This can be done by placing the symbol FAR after GO.

Example:

```
                                        MAC equivalents:

    GO FAR LL                               JMP I (LL
```

See also the GOSW operator in Sections 3.2.1 and 3.2.1.4.

3.2.5.2 SUBROUTINE CALLS

The simplest form of a subroutine call is the symbol CALL followed by a subroutine entry point, a pointer or a local label. If it is not yet defined, it is assumed to be an entry point of a succeeding subroutine. The parameters can be transferred by the registers.

The simple CALL statement can also be followed by a parameter list, being equivalent to the data list of the DATA statement. The compiler lays out the parameter addresses after the subroutine jump.

Examples:

```
                                        MAC equivalents:

INTEGER POINTER PNTR:=SUB0              PNTR, SUB0
CALL SUB1                               JPL I (SUB1
CALL PNTR                               JPL I PNTR
CALL SUB2 (V1, SX2, WM)                 JPL I (SUB2
                                        V1
                                        SX2
                                        WM
```

Parameters may also be transmitted, for instance, through a global base pointer:

```
NORD PL                                 MAC equivalents:

BASE BA                                 BA = * +200
INTEGER POINTER GP                      GP,0
ESAB
:
:
:
SUBR MAIN
INTEGER PAR1,PAR2                       PAR1,0; PAR2,0
MAIN: "BA" = :B                 MAIN,   LDA (BA
:                                       COPY SA DB
:
"PAR1"=:"GP"                            LDA (PAR1
CALL RUT                                STA GP–BA,B
:                                       JPL I (RUT
:
"PAR2"=:"GP"                            LDA (PAR2
CALL RUT                                STA GP–BA,B
:                                       JPL I (RUT
:
:                                   )FILL
RBUS                                )KILL PAR1, PAR2
RUT:SUBR RUT
GP + 3 * 4 = :GP                RUT,    LDA I GP–BA,B
EXIT                                    AAA 3
RBUS                                    MPY (4
                                        STA I GP–BA,B
                                        EXIT
```

3.2.5.3 SUBROUTINE EXITS

Return from subroutines can be performed by EXIT (which is compiled to EXIT) or EXITA (which is compiled to EXIT AD1). If these means for subroutine return are used, the programmer should insure that the L register contents have not be destroyed, for instance, by a subroutine call within the subroutine. When parameters are given in the CALL statement, he should also insure that the L register is correspondingly increased before an EXIT statement is given.

3.2.5.4 CONDITIONAL BRANCH STATEMENTS

The IF statement has the general form:

IF <conditions> THEN <statements> [ELSE <statements>] FI

The ELSE part may be omitted.

Between IF and THEN there may be several conditions delimited by the OR and AND symbols. The conditions are evaluated from left to right. If a condition followed by AND or THEN is not true, the statements after THEN are bypassed. If a condition followed by OR or THEN is true, the statements after THEN are executed. Otherwise, more conditions will be tested.

There are two types of conditions: *relations and bit tests.*

A *relation* consists of two executable expressions with a relational operator between them:

>	Greater	
<	Less	
=	Equal	
>=	Greater or Equal	
<=	Less or Equal	
><	Not Equal	
>>	Absolute greater	Not on NORD-1
<<	Absolute less	
>>=	Absolute greater or equal	
<<=	Absolute less or equal	

Example:

```
IF A<VAR OR VAR2>=VAR3 AND X> <A, THEN GO RET ELSE
A=:B FI
```

If the first element of the first expression in a relation is a pointer, a variable or a constant, the A, AD or TAD register is considered as the primary register.

If the first element of the second expression in a relation is an integer pointer, an integer variable or a constant, the T register is considered as the primary register. This means that the relation:

IF VAR1=VAR2 THEN . . . . is equivalent to

IF A:=VAR1=T:=VAR2 THEN . . . .

both VAR1 and VAR2 should be integers or integer pointers.

In a relation, the two expressions are, normally, evaluated and loaded into the A and T register before the actual comparison is made according to the given relation. The exception is when the second expression is equal to zero, i.e., in this case, the T register is not used (example below).

An expression may be empty. If it is, the present value of the A or T register will be used.

IF > THEN ... is equivalent to
IF A> THEN...

This means that a construction like this is possible.

IF VAR1=4 OR=6 OR=7 THEN... which is equivalent to
IF A:=VAR1=4 OR A=6 OR A=7 THEN...

If the first expression is of type real, the second must be equal to zero because, in this case, the TAD register is used for the first expression.

It is also possible to compare *absolute values*, by a subtraction and carry test. Then, the two relational operators may be used:

<< Absolute less
>>= Absolute greater or equal

In these two cases, one of the expressions should be equal to zero if a carry test is wanted. On NORD-1, this is the only way to compare absolute values.

Examples:

NORD PL

IF VAR–D<VAR2 THEN...

MAC equivalents:

```
LDA VAR
RSUB SD DA
LDT VAR2
SKP IF DA LST ST
JMP BYPAS
```

NORD PL	MAC equivalents:
IF A + 10 = 0 THEN...	AAA 10 JAF BYPAS
IF A – LLIM > > = 0 THEN	SUB LLIM BSKP ONE SSC JMP BYPAS (NORD-1 version)

The purpose of the last example is to examine whether the absolute value of A is greater than or equal to the absolute value of LLIM or not!

A condition may be a *bit test.*

If the bit to be tested is one of the single bit registers, then the test is performed to check whether this bit is equal to 1. The condition may be inverted by placing the symbol NBIT after the register specification.

Examples:

NORD PL	MAC equivalents:
IF K THEN....	BSKP ONE SSK JMP BYPAS
IF M NBIT THEN....	BSKP ZRO SSM JMP BYPAS

A bit in the general registers can also be specified. An expression determines the register. Then one of the symbols BIT or NBIT selects 1 or 0 as true. At last a constant determines the bit number.

Example:

	MAC equivalents:
IF T BIT 7 THEN....	BSKP ONE 70 DT JMP BYPAS
IF NBIT 1 THEN	BSKP ZRO 10 DA JMP BYPAS

The construction THEN GO <label> FI may be abbreviated to GO <label>.

For instance,

IF A<0 GO ERR

is equivalent to IF A<0 THEN GO ERR FI

MAC equivalent: JAN ERR.

3.2.5.5 UNCONDITIONAL LOOP CONTROL

The FOR statement is used for iterative purposes. Between FOR and DO are the iteration specifications. Between DO and OD are the statements to be executed.

The general version has the form:

FOR <control expression> [[STEP<step count>] TO <limit expression>] DO <statements> OD

If the <control expression> explicitly specifies a primary register, this register will be used as control register for the loop counting. I.e., the programmer must himself save the contents of that register at the beginning of the loop if he wants to use it for other purposes, and then load it again at the end of the loop.

If the <control expression> starts with a constant, the A register will be taken as primary register for the expression and used as control register for the loop counting.

If the <control expression> starts with an integer variable, the A register will be taken as primary register for the execution of this expression, but the integer variable will be used as control variable for the loop counting. I.e., the A register may be used for other purposes within the loop. Note, however, that the A register is loaded again with the contents of the integer variable and increased by the step count at the end of the loop so that the "user contents" are lost from one loop execution to another.

The <step count> is a constant defining the step size. STEP <step count> may be omitted, then assuming 1 as a default step.

If the <limit expression> does not specify a primary register, the T register will be the primary register. In any case, if a <limit expression> is specified it will always be evaluated again and compared to the control value BEFORE each execution. If the limit is exceeded, there will be no more executions.

Example:

FOR X:=VAR STEP 3 TO 50 DO A + ARR(X) OD

MAC equivalent:

```
        LDX VAR                 %SET INITIAL VALUE
NEXT,   SAT 50
        SKP IF DT GRE SX        %TEST LIMIT
        JMP BYPAS
        ADD I (ARR,X
        AAX 3                   %STEP CONTROL VAR
        JMP NEXT
BYPAS,  .....
```

Example:

MAC equivalents:

```
FOR J TO T DO CALL INCR OD      LDA J
                          NEXT, SKP IF DT GRE SA
                                JMP BYPAS
                                JPL I (INCR
                                LDA J
                                AAA 1
                                STA J
                                JMP NEXT
                          BYPAS,.....
```

The statement part

STEP <step count> TO <limit expression>

may only be omitted in two special cases. In both cases the loop test is exceptionally placed at the end of the loop:

1. A single variable between FOR and DO:

   Example:

   MAC equivalents:

   ```
   FOR VAR DO.....OD      NEXT,
                              MIN VAR
                              JMP NEXT
   ```

   This means that if the control variable contains a negative number before entering FOR, this will be the number of executions. The loop will in this case be executed at least once.

2. An X-expression between FOR and DO:

   Example:

   MAC equivalents:

   ```
   FOR X:=–5 DO.....OD        SAX –5 %5 EXECUTIONS
                          NEXT,
                              JNC NEXT
   ```

   The loop will, in this case, be executed at least once.

The loop can also start with just a single DO. Then there will be an unconditional jump back.

Example:

```
                                        MAC equivalents:

DO..... OD                              NEXT,

                                            JMP NEXT
```

In this case, there is no loop test at all and the loop must be left by means of an IF statement or a GOSW statement somewhere within the loop.

Example:

Two similar methods for writing loops:

```
FOR A STEP 2 TO T DO.....OD
```

is equivalent to

```
FORLO:  IF A < = T THEN
        —
        —
        —
        —
        A + 2
        GO FORLO
        FI;
```

3.2.5.6 CONDITIONAL LOOP CONTROL

The WHILE statement has the general form:

```
DO { <statements> WHILE <conditions>
   { <statements> WHILE <conditions> <statements> } OD
   { WHILE <conditions> <statements>
```

<conditions> may be *relations* or *bit tests* and must have the same syntax as in the IF statement. In a relation, the A and T registers are normally used to perform the comparison.

At least one of the two <statements> parts should be included. Each time the WHILE <conditions> are reached they are evaluated. If the result is "false" a jump to the next statement after OD is performed. If the result is "true", the <statements> between WHILE and OD are executed and a jump back to the beginning of the DO loop is performed.

Example:

```
DO WHILE VAR1>VAR2 AND X> <0
—
—
—
—
OD
```

In general, the WHILE statement may be placed anywhere within a DO loop. The code generated from a WHILE statement will always contain a conditional jump to the statement following the nearest enclosing DO-OD.

If one, for example, wants to test at the end of the loop, one could say:

```
DO
—
—
—
—
—
WHILE VAR=D OD
```

More complex expressions like the following are also legal:

```
FOR VAR1 STEP2 TO VAR2 DO WHILE X> <0
—
—
—
—
—
OD
```

EXERCISES

1. Let I1 and I2 be integers, D1 a double and R1 and R2 reals. Find the *primary operand* in each of the following expressions:

```
A:=I1+I2
I2+5=:I1
X:=I1
A:=X+I2
I1=:T
D1
R1+R2=:R2
```

2. Let V1 and V2 be (local) integers. Find the MAC equivalents of the following expressions:

```
V2+1=:V1
V1:=V2+1
V2:=:L
```

3. Let I1 and I2 be integer variables. Which ones of the following statements are *legal?*

```
T:=5
T:=+5
X:=–5
A:=:I1
I1:=:B
B:=:L
I2
+I2
A+I2
A:=+I2
5
+5
–5
–I1
A–I1
X–5
X–I1
A:=–I1
A:=I1–I2
```

```
T:=I1–I2
T:=–I2
A:=L*5
A:=5*L
A:=X*B
SHR  2
D  SHR  2
```

4. Let JJ be declared as a symbol. Which ones of the following statements are *legal?*

```
JJ
–JJ
+JJ
X–JJ
T–JJ
L–JJ
B–JJ
P–JJ
D+JJ
```

5. Let I1 be an integer, D1 a double and R1 a real variable. Which ones of the following statements are *legal?*

```
A:=I1
A=:D1
A:=R1
AD=:I1
AD:=D1
AD=:R1
TAD:=I1
TAD=:D1
TAD:=R1
T=:D1
T:=R1
X=:D1
X:=R1
AD:=D1+I1
TAD:=R1+D1
AD:=5
TAD:=5
TAD:=3.14
3.14=:R1
```

6. What is the ASCII code of the characters 0, 1, 2, ..., 9?
Study Appendix C!

7. Compile the following NPL program on your NORD–10 configuration and study the MAC output listing. Can you make the program more simple?

```
%N-PL PROGRAM READING ONE INPUT CHARACTER FROM
%THE TERMINAL AND WRITING THE NEXT ASCII CHARACTER ONTO
%THE SAME TERMINAL.  THE PROGRAM TERMINATES WHEN 0 IS READ.
SUBR SUB
INTEGER INT1, INT2
SUB:    T:=1;   *MON 1; MON 65          %INPUT ONE CHARACTER
        A–60=:INT1                      %REDUCE ASCII CODE TO BINARY
        IF A=0 THEN GO STOPP FI         %LAST CHARACTER?
        CALL SUB2(INT1)
        A:=15; T:=1;   *MON 2; MON 65   %OUTPUT CARRIAGE RETURN
        A:=12;  * MON 2;  MON 65        %OUTPUT LINE FEED
        INT1+60;   *MON 2;  MON 65      %BINARY NUMBER TO ASCII.
                                        %OUTPUT ONE BYTE
        GO SUB
STOPP:  * MON 0                         %RETURN TO SINTRAN III
RBUS

SUBR SUB2
INTEGER POINTER HOME
DISP 0; INTEGER D0; PSID
SUB2:   A:=L+1=:"HOME"                  %SAVE RETURN ADDRESS
        A:=L.D0.D0+1=:X.D0              %ADD ONE TO PARAMETER
        GO HOME                         %RETURN JUMP
RBUS

@ EOF
```

HINT: At least five statements may be changed!

Why is L+1=:"HOME" necessary instead of simply L=:"HOME"?

8. Write a NPL program which is reading 10 characters from the terminal into an array, sorting the characters in the opposite order and writing them out on the terminal again on the next line. The program should consist of two subroutines: one input/output routine and one sorting routine.

9. Given the integer number N. Consider the problem of calculating

$$N! = N*(N-1)*(N-2)..... *1 .$$

Write three small programs each reading the number N from the terminal and printing out N! on the next line.

The programs should utilize

i. the IF statement
ii. the FOR statement
iii. the WHILE statement.

10. Write a program which finds and prints out all the prime numbers smaller than 100 on the terminal.

HINT: Place all the integer numbers in an integer array of length 100. Then starting with N=2 clear all locations in the array containing multiples of N by writing 0 into these locations. Set N+1=:N and repeat the process. Continue until $N > \sqrt{100}$ (i.e., $N*N>100$). Print all the numbers in the array different from 0 on the terminal.

Array initially:

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17.....

Array after clearing:

1 2 3 0 5 0 7 0 0 0 11 0 13 0 0 0 17.....

11. Put the letters of your own name (in ASCII code) into an array initialized by a DATA statement. Write a program which prints the name on the terminal and then sorts all the letters into alphabetic order. The program should also count the number of As, Bs, Cs,.... etc. in your name and print the result on the terminal.

Example:

HABIBBOURGIBA
AABBBBGHIIORU

The name contains:

A	:	2
B	:	4
G	:	1
H	:	1
I	:	2
O	:	1
R	:	1
U	:	1

12. Which of the following NORD PL statement sequences are legal:

A)
```
INTEGER POINTER KP1
SUBR SU1
INTEGER K1
SU1: "K1"=: "KP1"
. . .
```

B)
```
INTEGER K1:=5
INTEGER POINTER KP1:=K1
SUBR SU1
SU1: A: = KP1
. . .
```

C)
```
SUBR SU1
INTEGER POINTER KP1: = K1
INTEGER K1
. . .
```

D)
```
SUBR SU1
REAL R1
DOUBLE POINTER DP1: = R1
. . .
```

E)
```
SUBR SU1
REAL R1
REAL POINTER RP1: = "R1"
. . .
```

F)
```
SUBR SU1
DOUBLE D1
INTEGER POINTER KP1
SU1: "D1"=:"KP1"
. . .
```

G)
```
SUBR SU1
INTEGER K1: = (1, 2, 3, 4)
. . .
```

H)
```
SUBR SU1
INTEGER K1, K2: = K1
```

13. Which of the following NORD PL statement sequences are legal:

A)
```
BASE BA
INTEGER ARRAY IA2 (100)
ESAB
SYMBOL S1 = BA
. . .
```

B)
```
BASE BA
INTEGER ARRAY IA3 (100)
ESAB
SYMBOL S1 = "BA"
. . .
```

C)
```
BASE BA
INTEGER POINTER IPB
ESAB
SUBR SU2
INTEGER I2
SU2:   "BA" =:B
       "I2" =:"IPB"
       X:=IPB
. . .
```

D)
```
SUBR SU1
BASE BA
INTEGER ARRAY IAB (100)
ESAB
DISP –200
INTEGER ARRAY IAD (100)
PSID
INTEGER ARRAY POINTER IP:=IAB
SU1:   "BA" =:B
       FOR X:=0 TO 77 DO
            IAB (X) + IP (X) =:IAD (X)
       OD
. . .
```

E)
```
SUBR SU1
DISP
INTEGER K1:=5
PSID
. . .
```

14. Which of the following NORD PL statement sequences are legal and reasonable:

A)
```
SUBR SU1
INTEGER ARRAY IA (100)
SU1: FOR T:=0 TO 77 DO 0=:IA(T) OD
. . .
```

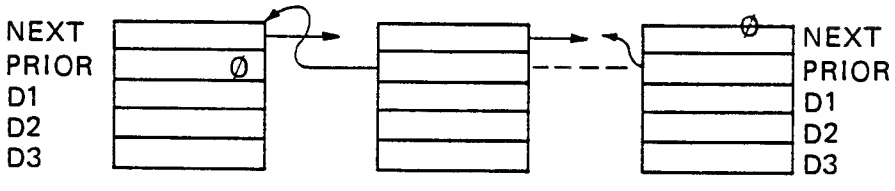
B)
```
IF K BIT THEN K:="0" FI
```

C)
```
IF K >< M THEN K:=M FI
```

D)
```
SUBR SU1
INTEGER K1:="0", K2:=0
INTEGER ARRAY KA (100), KB (100)
SU1:   FOR K1 TO 77 DO
           FOR K2 TO K1 DO
                   KA (K1) + KB (K2) =:KA (K1)
           OD
           0=:K2
       OD
. . .
```

E)
```
SUBR SU1
INTEGER ARRAY KA (100)
INTEGER K1:="0"
SU1:   T:=77
       FOR K1 TO T DO
           WHILE KA (K1) >< 0
           A + 1 =:KA (X)
       OD
. . .
```

15. Consider the following list structure:



Write a subroutine COUNT which counts the (original) number of elements in the list. The subroutine should also remove the first and last elements of the list.

Then add the integers D3 of all the elements in the list and place the sum into the parameter SUM given in the calling statement CALL COUNT (SUM).

Then add the integers D3 and D2 in each element and place the result into D1 of each element.

The A register is pointing to the first location of the first element in the list in the calling moment.

Hint: Utilize a DISP field!

16. Consider the two vectors VECT1 and VECT2 given in a global BASE field as follows:

```
BASE CAL
INTEGER ARRAY VECT1 (0)
DATA (1, 2, 3, 4, 5)
INTEGER ARRAY VECT2 (0)
DATA (0, –1, 0, –1, 0)
ESAB
```

Write a subroutine MULT1 which performs a vector-vector multiplication and places the result into the local integer RES.

17. Consider the matrix MAT and the vector VECT given in a global BASE field as follows:

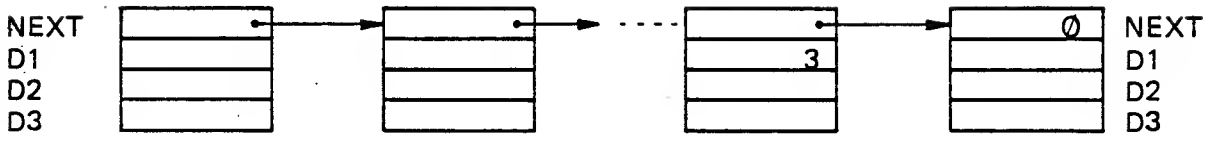
```
BASE CALC
INTEGER ARRAY MAT (0)
DATA (1, 2, 3, 4, 5)
DATA (6, 7, 8, 9, 10)
DATA (11, 12, 13, 14, 15)
DATA (16, 17, 18, 19, 20)
INTEGER ARRAY VECT:= (0, –1, 0, –1, 0)
ESAB
```

Write a subroutine MULT2 which performs a matrix-vector multiplication and places the result into the local vector RES declared as:

```
INTEGER ARRAY RES (4).
```

Output the result on the terminal.

18. Consider the following list structure:



Each element in the list contains the four integers NEXT, D1, D2 and D3.

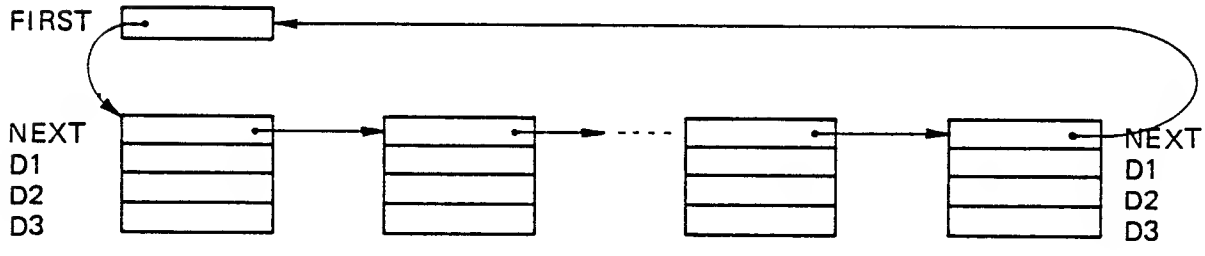
Write a subroutine REMOV which removes all elements whose D1 = 3 from the list.

The subroutine shall be called by the statement CALL REMOV and the A register shall contain the starting address of the first element of the list in the calling moment. If the A register is zero when entering REMOV, there are no elements in the list.

At return to the calling program, the A register shall point to the first element of the updated list structure or contain zero if all elements have been removed.

Hint: Utilize a DISP field.

19. Consider the following list structure:



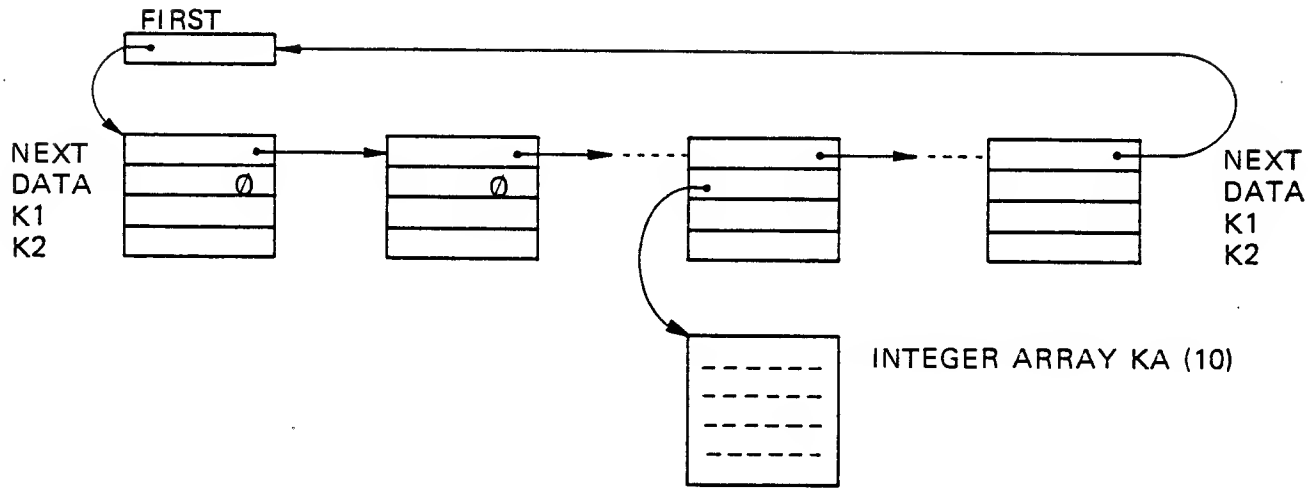
The integer pointer FIRST is pointing to a list of data elements each containing the four integers NEXT, D1, D2 and D3.

Write a subroutine SORT which sorts the data elements in ascending order by D1.

The subroutine shall be called by the statement CALL SORT (FIRST). FIRST is declared as an integer pointer in the calling routine and points to the first element of the list. NEXT of the last element points to FIRST.

If FIRST contains zero, there are no elements in the list.

20. Consider the following list structure:



The integer pointer FIRST is pointing to a list of data elements each containing 4 integers NEXT, DATA, K1 and K2. NEXT contains the address of the next element in the list, except for the last element where NEXT points to FIRST. DATA is either zero or contains the address of a data record starting with the integer array KA (10).

Write a subroutine SUM which adds the contents of the locations of each array KA and puts the sum into K1 of the corresponding data element.

The subroutine shall be called by the statement CALL SUM (FIRST) and FIRST is a local integer pointer in the calling program.

Hint: Utilize two DISP fields or one DISP field and an integer array pointer. Put special attention to the B and X registers! Remember that indexed variables may not appear in an expression using X relative addressing.

4 REAL TIME PROGRAMS

Real time programs (hereafter called RT programs) written in NORD PL must be compiled and assembled into units in binary relocationg format in order to be loaded by the real time loader.

This implies that the program units must be surrounded by the MAC commands)9BEG and)9END. The starting point (first instruction to be executed) of an RT program must be immediately preceded by the MAC command)9RT.

If a program unit refers to another program unit the MAC commands)9EXT and)9ENT must be present in the calling unit and in the unit called upon respectively.

If the)9RT command is used to declare a program unit as an RT program (main program), the command)9ENT shall not be used for the entry point (starting point) defined in the)9RT command.

After all program units the MAC command)9EOF must appear before the NORD PL command @EOF. The)9EOF command will cause an end-of-file mark to be written to the object file by the MAC assembler, thus, terminating a sequence of BRF program units.

All MAC commands must, of course, be preceded by an asterisk (*).

If the programmer wants to perform monitor calls from NORD PL, he must choose whether to establish his own library routines and insert them into his RT programs utilizing the @LIB and @ELIB commands, or whether he shall simply use the MAC monitor calls preceded by an asterisk.

How to establish RT programs in NORD PL is clearly demonstrated in the example in Section 5.1.

A list of the necessary MAC commands is found in Appendix D.

5 REENTRANT SUBROUTINES

5.1 *SUBROUTINES CALLABLE FROM MORE RT PROGRAMS*

NORD PL subroutines, which shall be parts of a real time program system and be simultaneously callable from more than one RT program, must be written in reentrant form. There are many ways to do this. In this section, two methods will be mentioned.

1. One method is described in Section 7.3 of the manual "SINTRAN III – Users Guide (ND-60.050)". The point here is to call the subroutine with an array, which is local to the calling program, as parameter. This array should then be used as an area for all data in the reentrant subroutine. In the reentrant subroutine this array should find its complement in a corresponding array declared in a DISP-field.

   Consider the following example:

```
%
SYMBOL PRIOR = 37
*)9BEG
*)9EXT ADD
SUBR MAIN
INTEGER ARRAY ARR(10)
*)9RT MAIN PRIOR
MAIN: FOR X:=0 TO 7 DO 0=:ARR(X) OD
      CALL SU1 (ARR)
      *MON 0
RBUS
*)9END
%
*)9BEG
*)9ENT SU1
SUBR SU1
DISP 0
INTEGER ARRAY BRR (10), ID = BRR
PSID
SU1:  L.ID =:B
      FOR X:=0 TO 7 DO BRR (X) + 1 =:BRR (X) OD
      EXITA
RBUS
*)9END
```

   The main program MAIN will initialize the array ARR with zeros and call subroutine SU1. This subroutine will load the address of ARR to the B register and increase the contents of the array with 1.

Subroutine SU1 uses no local variables, only the parameter ARR and the registers.

Thus, this subroutine is reentrant and may be called simultaneously from another RT program.

2. A second method is to use a BASE-field (or a normal field of variables) in the calling program as data area for the reentrant subroutine. The starting address of the BASE-field is transferred to the reentrant subroutine through the B register and this subroutine is accessing the BASE-field by means of corresponding DISP variables.

   Consider the example shown on page 5-3.

Example:

```
%
SYMBOL PRIOA=30,PRIOB=35
*)9BEG
*)9EXT PLUS
SUBR MAINA
BASE BA
     INTEGER A1, A2, A3
ESAB
*)9RT MAINA PRIOA
MAINA:    "BA"=:B
          1=:A1=:T
          2=:A2=:D
          "A1"=:B
          CALL PLUS (1)
          *MON 0
RBUS
*9END
%
*)9BEG
*)9EXT PLUS
SUBR MAINB
INTEGER B1, B2, B3
*)9RT MAINB PRIOB
MAINB:    3=:B1=:T
          4=:B2=:D
          "B1"=:B
          CALL PLUS (0)
          *MON 0
RBUS
*)9END
%
*)9BEG
*)9EXT MAINB
*)9ENT PLUS
SUBR PLUS
INTEGER MNB:=MAINB,PLIST:=MNB
DISP 0
     INTEGER P1, P2, P3
PSID
PLUS:     IF L.P1=0 GO ADD
          "PLIST"
          *MON 100
ADD:      P1+P2=:P3
          EXITA
RBUS
*)9END
%
*)9EOF
@EOF
```

The two RT programs MAINA and MAINB are both calling the same re-entrant subroutine PLUS. (The main programs and the subroutine are compiled and assembled into three different units in binary relocating format and loaded by the RT loader into the same segment.)

The common subroutine PLUS makes a test on the parameter and starts the RT program MAINB if this parameter is not zero. It then adds two integer variables and stores the sum into a third integer variable. By only considering subroutine PLUS, nothing can be said about which variables are really accessed. This depends completely upon the setting of the B register.

Thus, when called from MAINA the BASE-field variables will be accessed, when called from MAINB the integer variables in this program will be accessed.

The local variables MNB and PLIST in PLUS are not changed by any operations and may, therefore, be allocated inside of PLUS.

There is no space reserved for storing away register values because the SINTRAN III operating system will take care of the register contents of RT programs.

Now, take a look at the RT descriptions by executing the SINTRAN III commands

@LIST-RT-DESCRIPTION MAINA and
@LIST-RT-DESCRIPTION MAINB.

Then start the first RT program by executing

@RT MAINA.

When MAINA has called PLUS the RT program MAINB will be started and executed because MAINB has a higher priority than MAINA. Thus, we are sure that subroutine PLUS is actually performed twice "at the same time".

If you are quick and lucky, you might be able to see that both programs are in the execution queue by executing the command

@LIST-EXEC-QUEUE.

When both programs have terminated, have a new look at the RT descriptions and control whether the register values are correct!

5.2 *RECURSIVE SUBROUTINES*

However, if a subroutine must be reentrant because it is recursive (calls upon itself) or it is simultaneously called more than once in a long chain of subroutines calling each other, and all subroutines belong to only one RT main program or to a background program, a stack mechanism might be recommended.

The following example demonstrates the method. The following code must be inserted into the program system somewhere at the beginning.

```
%
@MAC
)MCDEF      SDATA
            A=6
            ]

)MCDEF      DATA $PAR
)KILL A
            A=$PAR+1
            ]

)MCDEF      ENTER
            STD       I    (ASTCK
            COPY      SL   DA
            COPY      SB   DD
            SAB       A
            JPL       I    (SPUSH
            ]

)MCDEF      LEAVE
            SAA       -A
            JMP       I    (SPOP
)KILL A
            ]

)MCDEF      ISTCK
            STA       I    (ASTCK
            LDA       (STACK
            STA       I    (CSTCK
            LDA       I    (ASTCK
            ]
```

```
@
%
%DISPLACEMENT WITHIN THE STACK ELEMENT
%
DISP 0
    INTEGER   XREG, TREG, AREG, DREG, LREG, BREG
    DOUBLE    ADREG=AREG; LBREG=LREG
    REAL      TADREG=TREG
PSID
%
%STACK FOR TEMPORARY DATA
%
INTEGER ARRAY STACK (700), ESTCK (7)
DOUBLE ASTCK
INTEGER POINTER CSTCK:=STACK
%
%PUSH ROUTINE FOR THE MACRO ENTER
%
SUBR SPUSH
SPUSH: *STX I  CSTCK
        X:="CSTCK"
        AD=:X.LBREG
        B+X
        "ESTCK"
        IF A<B GO STOVFL
        X:=:B
        X=:"CSTCK"
        AD:=ASTCK
        TAD=:TADREG
        X:=XREG
        EXIT
STOVFL: GO STOVFL
RBUS
%
%POP ROUTINE FOR THE MACRO LEAVE
%
SUBR SPOP
SPOP:   A+"CSTCK"=:"CSTCK"
        B=:X
        AD:=LBREG
        A=:L;  D=:B
        TAD:=X.TADREG
        X:=X.XREG
        EXIT
RBUS
```

The macro ISTCK must not necessarily be a macro. It is only written as such for the sake of convenience. This macro initializes the stack pointer CSTCK and must be called only once at the beginning of the main program.

The macros SDATA and DATA define the length of the corresponding stack element.

The reentrant subroutine is now designed in the following way. The stack area STACK is used for storing away register contents and local data. If only space for register contents is needed, the macro SDATA should be called before ENTER at the beginning of the reentrant subroutine.

If space for local data is required as well, a corresponding DISP-field starting at the value 6 should be introduced before the macros DATA and ENTER are called. The parameter of the macro DATA must be the value of the last "word" reserved by this DISP-field.

The macro ENTER stores away the register contents in a stack element and updates the stack pointer. All registers retain their values except for the B-register, which will now point to the beginning of the stack element and act as a base for the DISP variables.

In any case, the reentrant subroutine should be terminated by a call for the macro LEAVE, which reloads the original register contents, including the value of the B-register, from the stack element and updates the stack pointer.

Now, append, for instance, the following code after the above programs:

```
%
SUBR MAIN
MAIN:   *ISTCK
        3=:B=:D;  T:=0=:X
        CALL SUBA
        *MON 0
RBUS
%
SUBR    SUBA
        *SDATA
SUBA:   *ENTER
        X+1=:XREG
        A+1+60=:D;  CALL OUTA
        A-60
        IF A> <0 THEN CALL SUBB FI
        ##A=:D; CALL OUTA
        *LEAVE
RBUS
%
```

```
%
SUBR    SUBB
DISP6
    INTEGER K1,K2
PSID
        *DATA K2
SUBB:   *ENTER
        A—2=:K1
        ##B=:K2
        A:=40;  CALL OUTB
        K1+60;  CALL OUTB
        A—60
        CALL SUBA
        K2;  CALL OUTB
        K1+60:  CALL OUTB
        TREG+1=:TREG
        *LEAVE
RBUS
%
SUBR    OUTA, OUTB
OUTA:   A:=12;  T:=1;  *MON2;  MON 65
        A:-15;  *MON2;  MON 65
        A:=D.
OUTB:   *MON 2;  MON 65
        EXIT
        RBUS
%
@EOF
```

The main program MAIN puts initial values into the registers and calls subroutine SUBA. SUBA adds 1 to the A register and calls subroutine SUBB if the A register is not zero. SUBB subtracts 2 and calls SUBA. The two subroutines will now continue to call each other until the A-register becomes zero in SUBA.

Each time they are called they push up a new element in the stack. When one of the routines is left, the stack is popped back. The X and T registers are increased in the stack elements each time SUBA and SUBB are called.

Just to make it easy to follow the interaction, some output statements are introduced.

It is left as an exercise for the reader to compile and assemble the programs. After the assembly the system may be started by the command MAIN! to the MAC assembler. The following output is then printed on the terminal:

```
MAIN!
4  2
3  1
2  0
1  /
0
AB/
AB0
AB1
AB2
A
@
```

Now, give the SINTRAN III command @STATUS and have a look at the register values!

NOTE: The stack principle may not be used in the demonstrated way if more RT programs are using a common stack because the code for pushing and popping in the stack is not protected (for instance, by semaphores). The above technique will not work if an RT program is interrupted by another with higher priority inside of the SPUSH or SPOP routines.

Two RT programs may use a common subroutine in the demonstrated way if they have one stack and a set of SPUSH and SPOP routines each.

6 COMMON DATA AREAS

The concept of common areas was originally defined in FORTRAN, however, for many reasons it may be interesting to establish a common connection between FORTRAN, NORD PL and MAC programs.

6.1 *DEFINITION OF A COMMON AREA*

Under the SINTRAN III operating system, a common area is a data area which is accessible from more than one BRF program unit, independant of whether this is a main program or a subprogram or whether the accessing code was written in FORTRAN, NORD PL or MAC.

Common data areas may be used for communication between RT programs or for storing away data which are common to two or more RT programs. In background mode, a common area is common to one main program and one or more subprograms.

In NORD PL a common area is defined by the MAC statement)9ASF which must appear within a BRF unit in order to be loaded by the RT loader or by the background loader.

The concept of common areas should not be confused with global data areas in NORD PL.

Example:

A common area is defined without any program units in the BRF unit:

```
)9BEG
CSIZE = 1000
)9ASF  CLAB1  CSIZE
)9END
```

In this case, it is mostly convenient to write the statements directly in MAC so that the NORD PL compilation can be avoided. In the above example, the common block with the label CLAB1 will allocate an area of 1000_8 words.

Example:

A common area is defined in a BRF unit also containing program units.

```
*)9BEG
SYMBOL   CSIZE=1000,PRIOR=35
*)9ASF   CLAB2  CSIZE
%
INTEGER  ARRAY  IARR (100)
%
SUBR     MAIN
*)9RT    MAIN   PRIOR
MAIN:    CALL   SUBA
         *MON   0
RBUS
%
SUBR     SUBA
SUBA:    EXIT
RBUS
%
*)9END
*)9EOF
@EOF
```

The common area with the label CLAB2 will allocate an area of 1000_8 words. The integer array IARR is a global area only accessible from MAIN and SUBA.

6.2 *ACCESS OF A COMMON AREA*

At the time of programming, compiling and assembly, the address of a common area is undefined.

The MAC statement)9ADS gets hold of this address at load time and puts it into the location where the statement is placed.

The *)9ADS statement has two symbolic arguments, the first is the name of the common label and the second is the displacement relative to the start of the common area.

There are, of course, many ways of utilizing this address in NORD PL. In the example below, two methods are demonstrated. The first method is to load the address to the B register and access the common area by means of a DISP-field. The second method is to put the address of a common area into an integer array pointer and access the area through this pointer.

In the example below the values of the common area CLAB1 are loaded to the A register, increased by 1 and stored to the common area CLAB2.

```
*)9BEG
SYMBOL   CSIZ1=1000
*)9ASF   CLAB1  CSIZ1
*)9END
%
*)9BEG
SYMBOL   CSIZ2=1000,PRIOR=35,DSPL=0
*)9ASF   CLAB2  CSIZ2
%
INTEGER  CADR1
*CADR1/  )9ADS   CLAB1   DSPL
DISP 0
    INTEGER  ARRAY    COM1 (1000)
PSID
%
SUBR     MAIN
INTEGER ARRAY   POINTER  COM2
*COM2/   )9ADS   CLAB2   DSPL
*)9RT    MAIN PRIOR
MAIN:    CADR1=:B
         T:=777
         FOR X:=0 TO T DO
             COM1 (X)+1=:COM2(X)
         OD
         *MON 0
RBUS
%
*)9END
*)9EOF
@EOF
```

7 ADDITIONAL FEATURES

7.1 *COMMANDS*

A command starts with a circled alpha (@) followed by the command name. The command names are not reserved symbols, so that the same symbol can be used for a command name as well as for a user variable. After the command name parameters may follow, separated by commas.

Some of the commands are used for conditional compiling, being described in Section 7.2. In Section 7.3 on-line assembly coding is treated. The remaining commands are described below.

@ICR — "Ignore carraige return" mode
This command is to be used if a statement should need several lines (especially declaration statements). The carriage return is treated as if it were a space.

@CR — "Carraige return" mode
After this command carriage return will have the same effect as the semicolon (;), so that it will terminate the current statement.

@EOF — "End of file"
This command is used for exit from the compiler to the operating system. The MAC command)LINE is output on the object device. The command will list the number of errors detected during the compilation of the communication device.

@CLEAR — Clear the symbol table of the compiler.

@OCT — All integer numbers will be treated as octal.

@DEC — Integer numbers will be treated as decimal, except for those preceded by the "&" sign.

@DEV — <input device>, <list device>, <object output device>
This command is used for setting device numbers for the compiler. If the list device = 0, the error messages will be printed on the output communication device, otherwise on the list device.

Example:

@DEV 4,5,3
@DEV 4,0,3

If list output and object output use the same device number, the object output will appear left adjusted and the source program will be listed 32 columns to the right. The source program will be preceded by "%" signs, so that the mix can be assembled.

Example:

@DEV 4,5,5

For the TSS and SINTRAN III version files and devices may be specified symbolically in the TSS notation. The necessary closing and opening of files will be done. Numeric and symbolic representation may be mixed in the same DEV command.

If a device is not specified at all, the old one will be used.

Examples:

@DEV T–R, 0, OBJECT FILE
@DEV INP–FILE, L–P
@DEV INP, L–P, L– P

@MODE <input communication device>, <output communication device>
The communication devices will be defined. Normally, they will be equal to 1.

@XREF This command will add line numbers and a cross reference list to the listing.

@FLO32 Set 32 bits floating point format.

@FLO48 Set 48 bits floating point format.

The compiler will automatically set the right floating point format according to the hardware it is run on. The commands @FLO32 and @FLO48 are therefore only necessary for cross-compilations.

7.2 *CONDITIONAL COMPILING*

The form of conditional compiling is conceptual somewhat similar to the "Library mode" of the MAC assembler. This means that this facility is especially well suited for extracting modules from a symbolic library.

A module which could be included is headed by the command

@LIB

followed by a logical expression of symbols. For each symbol the compiler maintains an "include" flag which is automatically set to "true" if the symbol is undefined, and reset to "false" when the symbol is defined. However, the programmer can also explicitly put the "include" flag on or off using the commands

@STLIB	<symbol>	Set the "library include" flag
@NSLIB	<symbol>	Reset the "library include" flag

The expression after @LIB may have the operators

/\	And
\/	Or
–,	Not

The expression is evaluated from left to right. If the resulting "include" value is true, the following module will be included, otherwise it will be skipped.

The module is terminated by the command

@ELIB.

The @LIB - @ELIBs can be nested. If a module is skipped, it is skipped until its corresponding @ELIB.

Example:

```
:
:
CALL SUB1
:
:
CALL SUB2
:
:
@LIB SUB1 \/              %INCLUDE THE FOLLOWING IF
                          %SUB1 OR SUB2 HAS BEEN
                          %REFERENCED
SUBR SUB1, SUB2
:
:
@ELIB                     %UP TO THIS POINT
```

7.3 *IN-LINE ASSEMBLY CODING*

There are two ways of including assembly coding:

1. If a statement starts with an asterisk (*), the rest of the line will be taken as assembly code, being copied to the object output stream.

2. The command

   @MAC

   switches the compiler to assembly mode. The test will pass unchanged to the output stream until an alpha sign (@) is found.

   Examples:

```
     *TRA OPR
@MAC
     BORA 170 DX
@
```

8 USING THE COMPILER

8.1 *PREPARING NORD PL PROGRAMS*

The compiler may be used as a separate system outputting MAC assembly code to a file or external device.

If on line return to the compiler is wanted, the program should be ended with the command

@DEV 1

giving the control back to the operator, who may start a new compilation. If it is the last part to be compiled, it should instead end with the command

@EOF %EXIT FROM COMPILER

NOTE: If the @EOF command is forgotten, the last buffer contents will not be written on the object file and the file will not be closed!

Example of program:

```
%START OF PROGRAM
INTEGER     B1,B2
SUBR        SUB1
SUB1:       33=:B1
            5=:B2
            EXIT
RBUS
@DEV 1
```

In case of absolute programs (not BRF), it is not necessary with any special heading; the program can start with normal statements.

If the resulting program should be output in BRF format, the pertinent MAC commands)9BEG,)9ENT and)9EXT should be inserted as assembly code.

Example:

```
%SUBROUTINE TO PRINT 2 CHARACTERS
*)9BEG
*)9ENT OUT2
*)9EXT OUTBT
SUBR OUT2
     INTEGER WORD
     INTEGER POINTER LINK
OUT2:   T:=L=:"LINK"
        A=:WORD SHZ -10                  %LEFT BYTE
        T:=5; CALL OUTBT                 %LINE PRINTER
        WORD / \ 377;T:=5;CALL OUTBT     %RIGHT BYTE
        GO LINK
RBUS
*)9END
@EOF
```

Note that when a new input file, list file, or object output file is specified in the @DEV command, the corresponding old file will be closed.

Example:

Consider two source files, SF1 and SF2, containing some NORD PL routines. SF1 ends with @DEV 1 and SF2 ends with @EOF.

The command

@DEV SF1, OBJ, OBJ

will compile the source file SF1 and place the list and object output on the file OBJ. When the compilation is finished, control will return to the communication device (terminal).

If the command @DEV SF2 is now given, the source file SF2 will be compiled and the list and object output will be appended to the file OBJ.

If the command @DEV SF2, OBJ, OBJ is given, the file OBJ will be rewinded and the list and object output from SF2 will be placed from the beginning of OBJ.

In both cases, control will return to the operating system when the compilation of SF2 is finished.

8.2 *COMPILING NORD PL PROGRAMS*

Under SINTRAN III and TSS:

The NORD PL compiler is fetched by using the command @NORD PL. It then writes the message NORD PL <version number>, waiting for input from the terminal. Then give the @DEV command to set the appropriate devices.

The @DEV command has the general form:

@DEV <input file> <list file> <object file>

Note that the object program, i.e., the output from the NORD PL compiler is a symbolic MAC program. This object program in turn should be the symbolic input program to a following MAC assembler run.

Example:

Suppose a symbolic NORD PL program is placed on a paper tape. A compilation with a list of the program is wanted on the line printer and the MAC object (output) program is wanted on the line printer and the MAC object (output) program is wanted on the paper tape punch (fast punch). This is done as follows:

```
@NORD PL

NORD PL 74.12.07
@DEV T–R, L–P, F–P
```

Example:

Suppose the symbolic NORD PL program is already written onto the file NPL1 with the QED processor. Before the program is executed a last check is wanted for debugging purposes, i.e., the symbolic NORD PL program itself together with the MAC object program is to be written on the terminal. This is done as follows:

```
@NORD PL

NORD PL 74.12.07
@DEV NPL1,1,1
```

Example:

Suppose the symbolic NORD PL program is written onto the file NPL1 with the QED processor. A compilation is wanted with a list of the program on the terminal and with the MAC object program on the file MAC1:

```
@NORD PL

NORD PL 74.12.07
@DEV NPL1,1,MAC1
```

If a list of the program is not wanted, write:

```
@DEV NPL1,,MAC1
```

If the file MAC1 is not created yet, write:

```
@DEV NPL1,1,"MAC1"
```

The possibility of an interactive communication between the NPL programmer and the compiler exists by utilizing the command @DEV 1,1,1.

Example:

Suppose the NORD PL programmer wants to check out the different statement types of the language. He wants to type his statements on the terminal, to look at the MAC interpretation at once or (which might happen) get the error messages immediately. This is done as follows:

```
@NORD PL

NORD PL 74.12.07
@DEV 1,1,1
```

The compiler will now react by printing a % character on the terminal and the programmer is free to type his NORD PL statements. After each CR the compiler will give the corresponding MAC interpretation or error message and print a new % character on the terminal.

As mentioned in Section 8.1, the program should be ended with the command @DEV 1,0,0 or @EOF.

8.3 *ASSEMBLING AND EXECUTING NORD PL PROGRAMS*

As mentioned in Section 8.2, the output from the NORD PL compiler is a symbolic MAC assembler program. This program in turn should be the symbolic input program to a following MAC assembler run.

Before the MAC assembler is called the file containing the symbolic MAC program should normally be opened by the SINTRAN III command @OPEN-FILE.

Before executing the program, i.e., before leaving the MAC assembler it is a good rule to find the absolute address of the symbolic external entry point used as the main starting point of the program. This is for instance, necessary when starting the execution of the program with the SINTRAN III command @GOTO. The @DUMP command also requires the absolute value of a restart address. These addresses are obtained after the assembly by typing the symbolic name followed by a colon.

Now the program may be started either under control of the MAC assembler or under control of SINTRAN III.

Execution under control of the MAC assembler:

Example:

Suppose the symbolic MAC program with the starting point START is placed on the file MAC1 by the NORD PL compiler:

```
@OPEN MAC1, RX
FILE NUMBER IS 000101
@MAC

101$
START: 040000  START!
```

The $ command will start the assembler run. The assembler will answer with carriage return and line feed when finished. The : command will return the absolute value of the given external entry point and the ! command will start the execution of the program at the given symbolic/ absolute address.

Execution under control of SINTRAN III:

Example:

```
@OPEN MAC1, RX
FILE NUMBER IS 000101
@MAC

101$
START : 040000  )9TSS
@GOTO  40000
```

The MAC assembler is left by typing)9TSS. Back in SINTRAN III the command @GOTO < absolute address of starting point > will start the execution at the given absolute address.

To avoid reassembling before each execution, the object program from the MAC assembler may be placed on a binary file. This may also be done either under control of the MAC assembler or under control of SINTRAN III.

Writing the binary object program on a file under control of the MAC assembler:

Example:

Suppose the symbolic MAC program with the symbolic starting point FIRST is placed on the file MAC2 by the NORD PL compiler. The object program from the MAC assembler is to be placed on the binary file ABS 1 and later to be read and started from there.

```
@MAC

)9ASSM MAC2,,ABS1:BIN
FIRST:xxxxxx   *:yyyyyy   40000<yyyyyy-1
)BPUN FIRST
)9TSS
@PLACE-BINARY ABS1:BIN

@GOTO xxxxxx
```

*Note that in this case it is not required to open the file MAC2 before entering the MAC assembler. The command)9ASSM will open the file and start the assembler run. *: will return the value of the MAC location counter, i.e., the uppermost address of the object program plus one. The < command establishes the limits of the memory area to be written on the binary file by the)BPUN command. In this case, the object program should be read and started by the SINTRAN III commands @PLACE-BINARY and @GOTO. These two commands may be replaced by the @LOAD-BINARY command.*

Writing the binary object program on a file under control of SINTRAN III:

Example:

Suppose the symbolic MAC program with the symbolic starting point ENTRY and the reentry starting point REENT is placed on the file MAC3 by the NORD PL compiler. The object program from the MAC assembler is to be dumped to the file ABS3 and read and started from there.

```
@OPEN MAC3, RX
FILE NUMBER IS 000101
@MAC

101$
ENTRY:xxxxxx␣␣REENT'yyyyyy   *:zzzzzz
)9TSS
@MEMORY 40000␣zzzzzz–1

@DUMP ABS3:PROG
NUMBER:xxxxxx
NUMBER:yyyyyy

@RECOVER ABS3:PROG
```

The SINTRAN III command @MEMORY establishes the limits of the memory area to be dumped on the file ABS3 with the @DUMP command. The @RECOVER command will then read the program into core and start the execution.

For further information of the MAC assembler commands and of the SINTRAN III commands, see the manuals "Course Manual US01 MAC", "MAC User's Guide" and "SINTRAN III Users Guide".

A list of the most usual MAC commands is given in Appendix D.

8.4 *NORD PL LISTING WITH OCTAL ADDRESSES*

An octal address list may be obtained on the NORD PL source program listing by means of the)9SLPL command to the MAC assembler if the listing and the MAC object program have been put on the same file during the compilation.

Example:

Suppose a NORD PL source program is placed on the file INP by the QED processor. The following sequence of commands will give a listing of the NORD PL source program with the octal addresses corresponding to the first statement on each line. The listing will, in this example, be written on the line printer:

```
@NORD PL
NORD PL 74.12.07
@DEV  INP, MAC1, MAC1
–END OF COMPILATION
000000 ERRORS DETECTED

@MAC
)9SLPL
)9ASSM  MAC1, LINE-PRINTER,
```

If BRF object code is wanted, an object file name should be specified in the)9ASSM command.

8.5 *DIAGNOSTIC MESSAGES*

8.5.1 *Diagnostic Messages from the Compiler*

If the compiler detects an error, it prints a diagnostic message on the list device, preceded by some asterisks. If the list device is equal to zero, it prints, on the communication device, the name of the last label and the number of lines after the label, followed by the diagnostic message. Usually, the compilation will continue, however, in a few cases, the compilation has to stop, returning control to the operator (aborting if NORD-OPS).

Message	Meaning
Error, ill. base	Error in a BASE statement
Error, buffer full	Too long statement or object instruction.
Error in command	
Error in compiler	The compiler is destroyed, or may be there is a bug in the compiler.
Error, ill. condition	Error in the conditional compiling commands (LIB, SLIB, STLIB or NSLIB).
Error in data expression	Illegal operand or operator in a data expression.
Error in decl.	Error in a declaration statement.
Error, ill. disp.	Error in a DISP statement.
Error, ill. elem.	A basic element is found in a place where it should not be.
Error in elem.	An ill–formed basic element.
Error, in else/fi	Bad nesting of THEN–ELSE–FI
Error, ill. else/fi/od	Bad nesting of THEN–ELSE–FI or DO–OD
Error in expr.	Error in an executable expression.
Error in for	Error in a FOR statement.

Message	Meaning
Error in if	Error in an IF statement
Error in I/O	I/O error signalled by the surrounding system.
Error, no FI/OD	Unmatched THEN/ELSE or DO at the end of a subroutine.
Error, no (	Missing left parenthesis in a data list.
Error, ill. operation	This operation is not implemented in hardware, or non—corresponding operands.
Error in output	Error message from the surrounding system.
Error in relation	Ill—formed relation in an IF or FOR statement.
Ill. statement	The statement is illegal in this context, or illegal element in an expression.
Error in subr.	Error in a SUBR statement.
Error, table destroyed	Probably overlapping of compiled/ assembled program and the compiler's symbol table.
Error, table full	Too many symbols in the program.
Error, too complex	Too complex construction in an executable expression; the backtracking stack is filled.
Error, undefined	Undefined local symbols at the end of a subroutine.

8.5.2 *Diagnostic Messages from the Assembler*

Some errors can be detected at assembly time only, because the compiler does not keep track of memory address values. Following is a list of the most usual errors. For more information, see the manual "MAC User's Guide".

Message	Meaning
RANGE EX.	A label or variable is used too far away from where it was defined. It can for example occur for GO to a label defined earlier, or at a OD statement.
POSS.FLT	May be a label has been defined too far after the place where it was used. It can occur for a forward GO or in an ELSE, FI or OD statement, However, this message can occur if an undefined symbol is part of a data expression. Then it can normally be ignored.
(ERROR	Too far between the filling in of literals. The compiler outputs a)FILL command at each RBUS statement. However, the programmer can put *)FILL commands in between.

APPENDIX A

OPERATORS AND RESERVED SYMBOLS

A.1 *NON–ALPHANUMERIC ELEMENTS*

Arithmetic Operators

:=	Load
=:	Store
:=:	Swap
–	Subtract
+	Add
*	Multiply
/	Divide
\	Byte separator (Data expressions only)

Logical Operators

/\	And
\/	Or
–,	One's complement
–	Two's complement

Relational Operators

>	Greater
<	Less
=	Equal
>=	Greater or equal
<=	Less or equal
><	Not equal
>>=	Absolute greater or equal
<<	Absolute less
>>	Absolute greater
<<=	Absolute less or equal

Delimiters

:	Label definition
;	Statement terminator
.	X-addressing indicator
( )	Array index or data list
"	Referenced variable or data expression
#	Character constants
%	Comment
&	Octal number
'	String
*	MAC instruction
@	Command
?	Undefined location

A.2 *RESERVED SYMBOLS*

Registers

A, X, T, B, L, D, P, AD, TAD
K, Z, Q, O, C, M

Declarations

INTEGER, DOUBLE, REAL, TRIPLE
ARRAY, POINTER, SYMBOL, DATA
BASE, ESAB
DISP, PSID
SUBR, RBUS

Statement Symbols

GO, CALL
IF, THEN, ELSE, FI, AND, OR
FOR, STEP, TO, DO, OD, WHILE
EXIT, EXITA, FAR

Operators

XOR
BONE, BZERO
SHZ, SH, SHR, SHL
GOSW
MIN
BIT, NBIT

APPENDIX B

PROGRAMMER'S CHECK LIST

i. Initjalize the B—register on program entry.

ii. Provide for exit from the subroutines.

iii. Put proper termination (@ DEV or @ EOF) at the end of the program.

iv. Check the @ICR – @CR pairs. Remember the semicolon after @ CR!

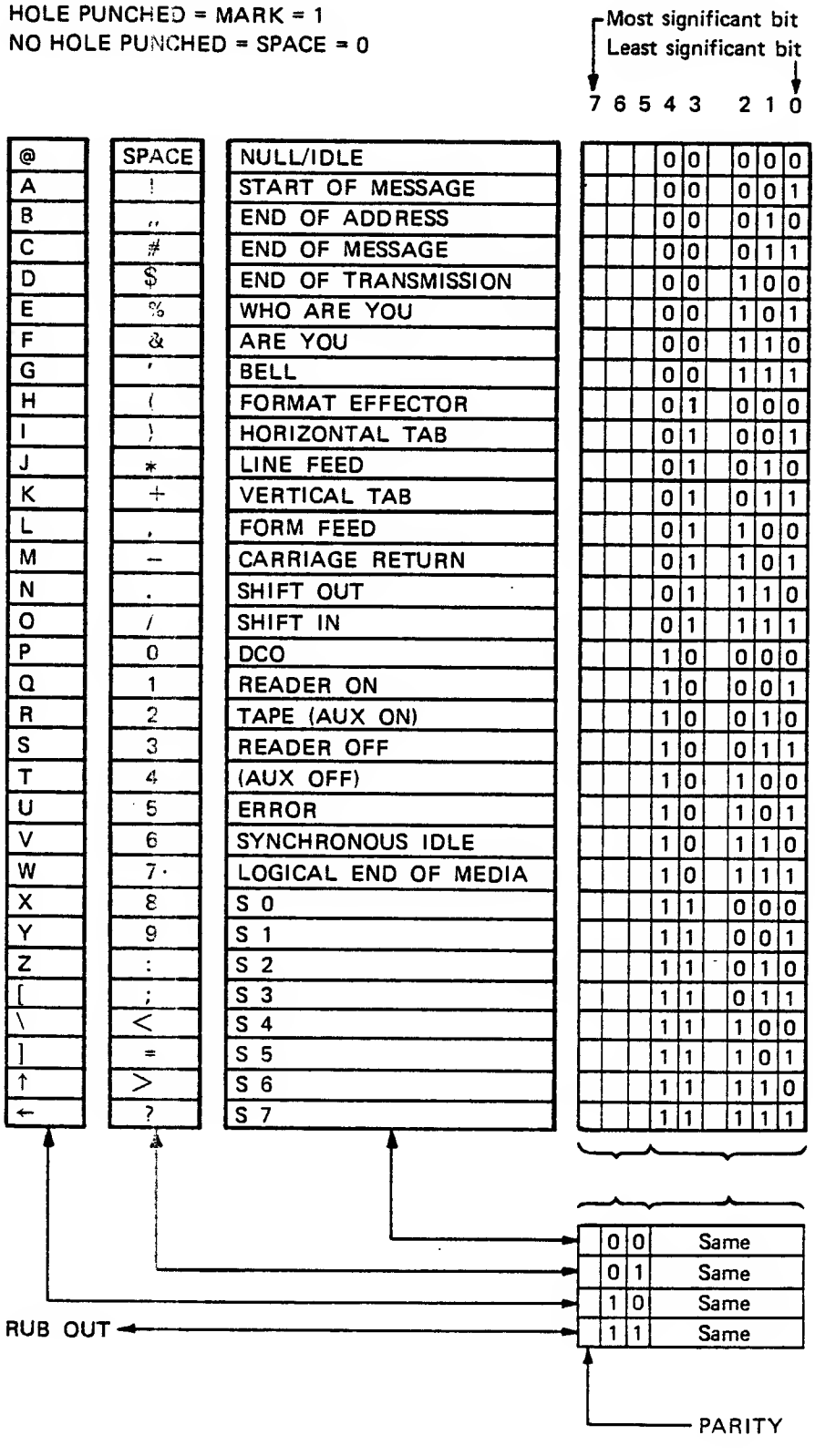
v. Check the IF – FI and DO – OD nestings.

vi. Check that the X relative addressing is remembered on all relevant places, e.g., that VAR is *not* written instead of X.VAR.

APPENDIX C

MODEL 33 ASR/KSR TELETYPE CODE (ASCII) IN BINARY FORM

HOLE PUNCHED = MARK = 1
NO HOLE PUNCHED = SPACE = 0



APPENDIX D

DEFINITION OF SOME MAC COMMANDS

Some of the commands to the MAC assembler are also useful for the assembling and loading of NORD PL programs. Among these commands are the following:

)9ADS <common block label> <displacement>
is used to access labelled common variables in a real time program. Blank common is referred to through the symbol 8COM. The <displacement> relative to the starting address of the common block must be separated from the <common block label> by a blank or a plus sign. The <displacement> must be a not relocatable identifier declared by means of the MAC statement = or by the NORD PL statement SYMBOL. At load time the address of the <common block label> is added to the <displacement> and put into the location where the)9ADS command appears.

)9ASF <common block label> <no. of words>
defines a common area with the name and size as specified. <no. of words> must be a not relocatable identifier.

)9ASSM <source file>, <list file>, <object file>
starts the assembly of a MAC program.

)9BEG [<label>]
puts the MAC assembler into a binary relocating mode, i.e., the object program unit will be in binary relocatable, linkable form. The object program may now be loaded by the NORD Relocating Loader or by the RT Loader.

<label> is the starting point and only to be specified if the program unit represents a background main program. Then the RUN command to the NORD Relocating Loader will start the program at this address. If the object device is a paper tape punch,)9BEG also causes 200 frames of blank paper tape to be output.

)9END resets MAC to produce an absolute output. This is the complement of the)9BEG command and must terminate each program unit starting with)9BEG. All identifiers defined since the last)9BEG are deleted from the symbol table. Identifiers referenced since the last)9BEG, but undefined, are printed on the list device. If the object device is a paper tape punch, 200 frames of blank tape is output. If MAC was already in absolute assembly mode when)9END was given, symbols defined since the last)9ENT command are deleted from the symbol table.

)9ENT <identifier 1> <identifier 2> . . .
declares symbols, variables and labels which may be referred to as external identifiers from other program units.

Thus,)9ENT is the complement of)9EXT. Delimiter is a space.

)9EOF will cause an end-of-file mark to be written to the object file, thus, terminating a sequence of BRF program units.

)9EXT <identifier 1> <identifier 2> . . .
declares identifiers to be external to the particular program unit being assembled. Identifiers declared with this command must appear in a)9ENT or a)9RT command of another program unit and a suitable linking will be made at load time.

)9RT <identifier> <priority>
declares a program unit to be an RT program with the name <identifier> and the specified <priority>. The last parameter must be a not relocatable identifier.

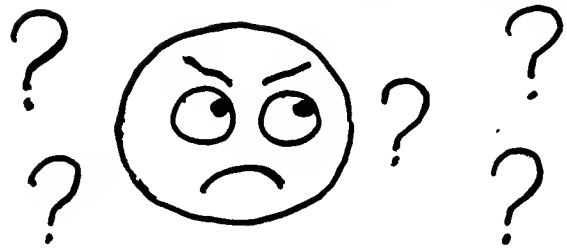
This command must appear immediately before the statement in the RT program where the label <identifier> is defined.

)9SLPL gives an octal address list of a NORD PL source program when given before the)9ASSM command. The listing of the NORD PL source program and the MAC object program must have been written onto the same file during the compilation.

APPENDIX E

ALPHABETICAL INDEX OF THE MANUAL

* * * * * * * * * * SEND US YOUR COMMENTS!!! * * * * * * * * * *



Are you frustrated because of unclear information in this manual? Do you have trouble finding things? Why don't you join the Reader's Club and send us a note? You will receive a membership card - and an answer to your comments.

Please let us know if you

* find errors
* cannot understand information
* cannot find information
* find needless information

Do you think we could improve the manual by rearranging the contents? You could also tell us if you like the manual!!

* * * * * * * * * HELP YOURSELF BY HELPING US!! * * * * * * * * *

Manual name: NORD PL-User's Guide Manual number: ND-60.047.03

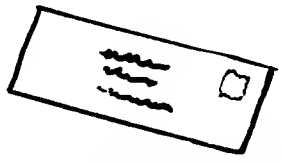
What problems do you have? (use extra pages if needed) _____

Do you have suggestions for improving this manual? _____

Your name: _____ Date: _____
Company: _____ Position: _____
Address: _____

What are you using this manual for? _____

Send to: Norsk Data A.S.
Documentation Department
P.O. Box 4, Lindeberg Gård
Oslo 10, Norway



Norsk Data's answer will be found on reverse side

Answer from Norsk Data: _____

Answered by: _____ Date: _____

Norsk Data A.S
Documentation Department
P.O. Box 25 BOGERUD
N - 0621 OSLO 6 - Norway